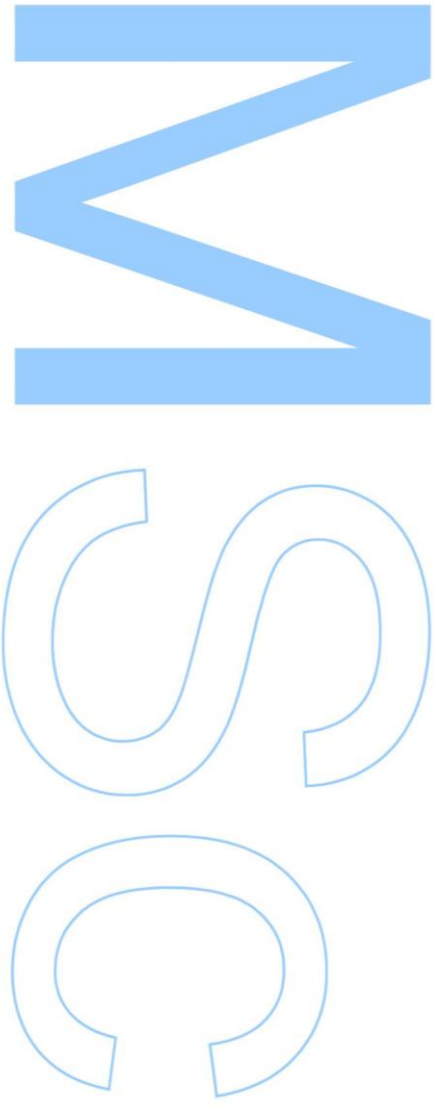
Implementação de um servidor de negociação em bolsa baseado em WebSocket

Vasco Morais Themudo
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2014

Orientador
Rui Silva, Colaborador, Finantech

Co-orientador
José Paulo Leal, Professor, FCUP



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Agradecimentos

O trabalho que apresento não teria sido possível realizar sem a ajuda e colaboração de várias pessoas.

Começo por agradecer a orientação e dedicação do professor José Paulo Leal, por todas as pertinentes questões que me foi colocando ao longo do estágio, bem como, todas as correções e sugestões de alteração ao relatório. Agradeço igualmente ao orientador (inicial) Hugo Oliveira, que criou todas as condições para o desenvolvimento do trabalho, definindo os principais conceitos e objetivos a atingir.

Agradeço a disponibilidade, o apoio e a boa disposição de Rui Silva e Nuno Teixeira, que me acompanharam mais de perto no desenvolvimento do protótipo.

Agradeço igualmente a João Marta da Cruz e Daniel Eusébio pelo acolhimento na Finantech e todo o acompanhamento e motivação ao longo do estágio.

Agradeço ainda a Pedro Silva e toda a equipa IT, Nélson Eusébio e Arthur Cunha, pela disponibilidade, prontidão e profissionalismo na resolução de problemas que surgiram esporadicamente.

A todos, muito obrigado.

Resumo

A diversidade de instrumentos financeiros, a monitorização rigorosa dos investimentos, a automatização de operações financeiras, os inúmeros requisitos de segurança, entre muitos outros factores, constituem atualmente um enorme desafio para os sistemas informáticos. Para além disso, a pluralidade de canais de atendimento e diferentes plataformas de acesso por parte dos investidores aos mercados financeiros, exige o desenvolvimento de aplicações Web robustas, rápidas e seguras.

O objectivo deste trabalho é desenvolver um protótipo de um sistema que permita diminuir a latência na comunicação e aumentar a coerência de dados entre o *ServerDealWeb* e todos os componentes a ele ligados. O *ServerDealWeb* é um servidor cujo principal objetivo é fornecer dados de mercado em tempo real a aplicações Web.

De forma a superar as limitações do modelo pedido/ resposta do protocolo HTTP são apresentadas as principais alternativas tecnológicas de comunicação para fornecer dados a aplicações Web em tempo real. Das várias soluções apresentadas destaca-se o protocolo *websocket*. Esta tecnologia, compatível com o HTML, permite a criação de um canal de comunicação bidirecional, persistente e em tempo real, sobre um único *socket.*

Os testes de performance comparativos entre o *ServerDealWeb* e o protótipo desenvolvido demonstram que a utilização do protocolo *websocket* permite uma utilização mais eficiente da largura de banda e um menor número de troca de mensagens para a mesma quantidade de informação.

No futuro pretende-se criar uma biblioteca DLL, portável para ambientes Windows e Linux, que possa ser incluída em qualquer projeto e atuar como cliente ou servidor, implementando o protocolo *websocket* para a comunicação. Também será necessário alterar todos os componentes que se ligam atualmente ao *ServerDealWeb*.

Abstract

The variety of financial instruments used nowadays, the automation of financial transactions and operations, the amount of financial security requirements, the permanent monitoring of investments, among other factors, introduces new challenges for computer systems. Besides, the number and heterogeneity of digital platforms used by investors to access financial markets requires the development of reliable, fast and secure Web applications.

The purpose of this work is to develop a prototype system to enable low latency communication and increase data consistency between *ServerDealWeb* and all components connected to it. This server provides real-time market data to Web applications.

In order to overcome the limitations of the HTTP model of request/ response, in this document we review the literature about the most significant technologies which provide data to web applications in real time. The *websocket* protocol is chosen to develop the prototype system. This technology, compatible with HTML, provides bi-directional, full-duplex, real-time, client/server communications over a single TCP connection.

The performance tests between the prototype system and *ServerDealWeb* show that the *websocket* protocol enables a more efficient use of bandwidth and a smaller number of messages exchanged for the same amount of data transferred.

In the future it's supposed to create a dynamic link library, portable to Windows and Linux, which can be included in any project and act as client or server, implementing the *websocket* protocol for communication. All components that are currently connected to *ServerDealWeb* should also be changed.

Índice

1. Introdução 1

1.1. Mercados Financeiros 1

1.2. Apresentação da Finantech 5

1.3. Apresentação do ServerDealWeb 6

1.4. Definição de objectivos 10

1.5. Evolução do trabalho 12

2. Tecnologias relacionadas 14

2.1. Introdução 14

2.2. Polling 16

2.2.1. Short polling 16

2.2.2. Long polling 17

2.3. BOSH 18

2.4. Streaming 18

2.4.1. Server-Sent Events 18

2.4.2. Forever frame 19

2.5. XMLHttpRequest 20

2.6. Web messaging 21

2.7. Web Real-Time Communication 22

2.8. SPDY 23

2.9. PubSubHubbub 24

3. WebSocket 25

3.1. HTML5 25

3.2. Protocolo websocket 28

3.2.1. Pedido inicial 28

3.2.2. Formato das mensagens 30

3.3. API websocket 32

3.3.1. Construtor 32

3.3.2. Eventos 33

3.3.3. Métodos 33

3.3.4. Atributos dos objetos websocket 34

3.4. Segurança 34

3.4.1. Analisar o cabeçalho: origin 35

3.4.2. Utilizar wss 36

3.4.3. Validação dos dados do cliente e servidor 37

3.4.4. Autenticação e autorização 38

3.4.5. Validação do subprotocolo 38

3.4.6. Limitar o número de ligações 38

3.4.7. Evitar comunicação via túnel 39

4. Apresentação do protótipo 40
4.1. Escolha da tecnologia 40
4.2. Implementação e arquitetura do software 41
4.2.1. Classe wsServer 42
4.2.2. Classe clientConnection 45
4.2.3. Classe ICom 47
4.2.4. Outras classes 47
4.2.5. Clientes Web 48
4.3. Protocolos e formatos auxiliares 49
4.3.1. Protocolo FIX 49
4.3.2. Formato JSON 51
4.4. Execução 52

5. Avaliação e testes de performance 54
5.1. SDW vs protótipo 54
5.2. Protótipo 57

6. Conclusão 60

7. Referências bibliográficas 63

Índice de figuras

Figura 1.1: Enquadramento do *ServerDealWeb* na plataforma Sifox 7
Figura 1.2: Ligação entre SDW e *SifoxDealWeb* e *SifoxDealMobile* 9
Figura 1.3: Ligações do SDW 10
Figura 1.4: Etapas de desenvolvimento do trabalho 12
Figura 2.1: Sessão HTTP 15
Figura 2.2: Formato de um endereço URL 15
Figura 2.3: *Short polling* 17
Figura 2.4: *Long polling* 17
Figura 2.5: Funcionamento do protocolo BOSH 18
Figura 2.6: HTTP Streaming 19
Figura 2.7: Pedido AJAX 21
Figura 2.8: Arquitetura WebRTC 22
Figura 2.9: HTTP vs SPDY 23
Figura 3.1: Servidor *websocket* 26
Figura 3.2: Etapas de uma ligação *websocket* 28
Figura 3.3: pedido inicial HTTP de cliente para ligação *websocket* 29
Figura 3.4: resposta HTTP do servidor 29
Figura 3.5: Mensagem *websocket* 30
Figura 3.6: Detalhes de uma mensagem *websocket* enviada por um servidor 31
Figura 3.7: Detalhes de uma mensagem *websocket* enviada por um cliente 31
Figura 3.8: HTTP, HTTPS, WS e WSS 37
Figura 4.1: Esquema geral do protótipo 42
Figura 4.2: Estrutura geral do SDW 43
Figura 4.3: Gráfico de fluxo da classe *wsServer* 44
Figura 4.4: Gráfico de fluxo da classe *clientConnection* 46
Figura 4.5: Cliente *websocket* - troca de mensagens 48
Figura 4.6: Cliente *websocket* que recebe mensagens do *ServerDeal* 49
Figura 4.7: Ligação *websocket* entre cliente e servidor 52
Figura 4.8: Início de ligação *websocket* entre cliente e servidor 53
Figura 5.1: Arquitetura para os testes com SDW 55
Figura 5.2: Comparação entre mensagens nos canais a e b, dos servidores para clientes 56
Figura 5.3: Uma mensagem *websocket* com três pacotes 56
Figura 5.4: Quantidade de tráfego gerado em média, por segundo, nos canais a e b 57
Figura 5.5: Monitorização da memória física e virtual durante a ligação de 400 clientes 58
Figura 5.6: Relação entre o número de clientes ligados e a quantidade de *threads* 58

Índice de tabelas

Tabela 3.1: Ataques ao protocolo e API *websocket* e respectivas medidas de segurança 39
Tabela 4.1: Lista dos clientes do SDW com suporte *websocket* nativo e não nativo 40
Tabela 5.1: Tráfego gerado nos canais a e b, durante 60 minutos 57
Tabela 5.2: Número de *threads* criados para 0, 1 e 2 ligações 58

Índice de listagens

Listagem 3.1: Geração da chave de resposta (*Sec-WebSocket-Key*) 30
Listagem 3.2: Construtor *websocket* 32
Listagem 3.3: Construtor *websocket* com suporte a protocolos 32
Listagem 3.4: Eventos *websocket* 33
Listagem 3.5: Métodos *websocket* 34
Listagem 4.1: Mensagem no formato JSON 51

Lista de abreviaturas

AJAX	—	*Asynchronous JavaScript and XML*
API	—	*Application Programming Interface*
BOSH	—	*Bidirectional-streams Over Synchronous HTTP*
CSFR	—	*Cross-site request forgery*
DoS	—	*Denial of Service*
FO	—	*FrontOffice*
HTTP	—	*Hypertext Transfer Protocol*
IETF	—	*Internet Engineering Task Force*
IIS	—	*Internet Information Services*
MITM	—	*Man in the Middle*
PSI-20	—	*Portuguese Stock Index*
RFC	—	*Request for Comments*
SDW	—	*ServerDealWeb*
SSE	—	*Server Sent Events*
TLS	—	*Transport Layer Security*
URI	—	*Uniform Resource Identifier*
URL	—	*Uniform Resource Locator*
W3C	—	*World Wide Web Consortium*
XHR	—	*XMLHttpRequest*
XMPP	—	*Extensible Messaging and Presence Protocol*
XSS	—	*Cross-site scripting*

1. Introdução

Este relatório foi realizado no âmbito do estágio do segundo ano de mestrado em Ciência de Computadores, da Faculdade de Ciências da Universidade do Porto, enquanto unidade curricular anual e obrigatória. O trabalho foi realizado na empresa Finantech — Sistemas de Informação, S.A. (Finantech), entre Novembro de 2013 e Junho de 2014. A Finantech é uma empresa de tecnologias de informação que se especializou na produção de soluções de *software* integradas para os mercados financeiros e de investimentos.

Este capítulo inclui: (i) uma introdução aos mercados financeiros, a sua evolução histórica até à atualidade, estrutura e modo de funcionamento e definição de alguns conceitos importantes; (ii) apresentação da Finantech e dos seus principais produtos que disponibiliza aos clientes; (iii) contextualização do *ServerDealWeb* (SDW) na plataforma Sifox; (iv) definição dos objectivos deste trabalho; e (v) caracterização das principais etapas percorridas durante o estágio.

1.1. Mercados Financeiros

Até à antiguidade clássica todas as transações eram realizadas através da simples troca direta de bens ou serviços (Goncalves, 1984). Com a evolução e complexidade das relações de troca surgiu a *moeda* que passa a permitir atribuir um determinado preço a bens e serviços, possibilitando a sua compra e venda. A criação da moeda como ferramenta de valoração veio facilitar a atividade económica, sobretudo pelo facto de permitir poupança e investimento (APB, 2014).

A *economia* tenta compreender a forma como as pessoas utilizam os recursos produtivos escassos — terra, capital e trabalho —, para atingir os seus objetivos. Podemos dividir a economia em microeconomia e macroeconomia (Vieira, 2004). Enquanto a primeira estuda o comportamento do indivíduo, a macroeconomia analisa a evolução da economia como um todo, avaliando, por exemplo, as variações do Produto Interno Bruto, da taxa de desemprego e da inflação (Martins, 2007).

É neste contexto que surgem os *mercados financeiros*. Um mercado financeiro é um mecanismo que permite direcionar os recursos excedentes da economia (poupança) para o financiamento de empresas e de novos projetos (investimentos). Ou seja, permitem efetuar a troca comercial (compra e venda) de: valores mobiliários (por exemplo, ações e obrigações), mercadorias (como pedras preciosas ou produtos agrícolas), câmbio de moedas e outros bens.

A finalidade dos mercados financeiros é a alocação eficiente da poupança entre aqueles que necessitam de recursos financeiros a curto prazo e os investidores, aqueles que dispõem de recursos financeiros e só pretendem usá-los a longo prazo. Este processo de alocação é realizado através dos seguintes intermediários financeiros: (i) instituições de crédito (bancos); (ii) empresas de investimento em valores mobiliários (sociedades corretoras, financeiras e gestoras de patrimónios); (iii) sociedades gestoras de instituições de investimento colectivo ou fundos de investimento (CMVM, 2011).

Os mercados financeiros englobam vários tipos de mercados (BM&F Bovespa, 2007):

- *Mercado monetário* — formado por bancos comerciais, múltiplos e sociedades de crédito (operações com moeda local);
- *Mercado de crédito* — onde são negociadas as operações de empréstimos, arrendamento mercantil e financiamentos para pessoas físicas e jurídicas;
- *Mercado de câmbio* — onde são realizadas as operações de compra e venda de moeda estrangeira com taxas flutuantes e livres;
- *Mercado de capitais* — onde são realizadas as operações de compra e venda de ações, títulos e valores mobiliários efetuados entre pessoas físicas e jurídicas.

É no mercado de capitais que as empresas que necessitam de capital conseguem financiamento, por meio da emissão de títulos (de dívida, obrigações ou ações) os quais são adquiridos pelos investidores (pessoas ou empresas). No entanto, não apenas esses investidores podem não querer ficar muito tempo com os títulos que adquiriram, como também é provável que existam outros investidores interessados em comprar esses títulos. Assim nasceu a ideia de se criarem as *bolsas de valores*, que representam o mercado onde os investidores compram e vendem títulos anteriormente emitidos por empresas (IBS, 2006).

Em Portugal as referências mais remotas relativas ao aparecimento das bolsas encontram-se na Idade Média (Santos, 2001). O desenvolvimento do comércio, por um lado, originou um maior contacto entre os comerciantes e, por outro lado, a presença frequente de negociantes estrangeiros atraía os corretores que, pelo facto de falarem várias línguas facilitavam as transações de mercadorias. Em 1495 surge a primeira tentativa de regular a atividade dos corretores de Lisboa e na segunda metade do século XVIII surgem as primeiras emissões de ações e títulos de dívida pública moderna. Só a partir de 1837 se consagra a existência de corretores de valores e se enquadra juridicamente a atividade comercial em geral, definindo-se conceitos como Praça de Comércio ou Bolsa de Valores. Em 1889 com a aprovação de um novo

Regulamento das Bolsas deu-se um impulso decisivo na sua criação, devido a dois factores: (i) são estabelecidas as regras de funcionamento das bolsas de valores e (ii) é construído o enquadramento regulamentar que previa a existência de segmentos nas bolsas para se negociarem valores mobiliários. Finalmente, em Janeiro de 1891 e Outubro de 1901, respectivamente, são criadas as Bolsas de Valores do Porto e de Lisboa.

Os principais serviços que as bolsas disponibilizam são: depósito de valores mobiliários; índices de mercado e de desempenho das bolsas de valor; informação económica e financeira de mercado; liquidação de operações de câmbio e de transferências livres e financeiras; registo de valores mobiliários; sistemas de gestão de empréstimos e de liquidação interbolsa (Euronext, 2013).

Desta forma as bolsas desempenham duas importantes funções. Por um lado, garantem aos investidores a possibilidade de transformar o seu investimento em dinheiro através da venda de títulos. Por outro lado permitem que qualquer investidor aplique em qualquer momento o capital que tenha disponível através da compra de títulos.

Os movimentos de uma bolsa de valores são obtidos através de índices, cuja variação reflete a tendência da bolsa. Existem vários processos para calcular o valor de um índice, nomeadamente, considerar as ações mais negociadas num determinado mercado, as de maior ou menor capitalização, ou de um sector específico. O método mais frequente considera o valor de mercado das empresas que compõem os índices. Exemplos dos principais índices a nível mundial: *Dow Jones* (EUA), o *Nasdaq100* (EUA), o *Ibex35* (Espanha), o *CAC* (França), o *FTSE* (Reino Unido), o *DAX* (Alemanha), o *Euronext100 e Euronext150* (Paris, Amsterdão, Bruxelas e Lisboa), o *Nikkei* (Japão) e o *Hang Seng* (Hong Kong). O PSI-20 (acrônimo de *Portuguese Stock Index*) é o principal índice de referência do mercado de capitais português. É composto pelo preço das ações das vinte maiores empresas cotadas na bolsa de valores de Lisboa.

Valores mobiliários são documentos emitidos por empresas ou outras entidades, em grande quantidade, que representam direitos e deveres, podendo ser comprados e vendidos. Para as empresas que os emitem, representam uma forma de financiamento alternativa ao crédito bancário. Para os investidores são um modo de aplicação de poupanças alternativo aos depósitos bancários e a outros produtos financeiros que se caracteriza por oferecer níveis diferentes de risco e rendibilidade.

Os valores mobiliários mais conhecidos são: ações, obrigações, unidades de participação em fundos de investimento, títulos de participação, futuros, opções, unidades de titularização de créditos, *warrants* autónomos, direitos destacados de

valores mobiliários, certificados, valores mobiliários obrigatoriamente convertíveis e por opção do emitente e valores mobiliários condicionados por eventos de crédito.

Em seguida são apresentadas as características dos principais valores mobiliários. As *ações* são títulos que representam uma fracção do capital social de uma empresa, constituída sob a forma de sociedade anónima. O detentor destes títulos adquire um conjunto de direitos sobre a sociedade. Estes direitos variam em função do número e da categoria das ações detidas.

O retorno de um investimento em ações depende da evolução do seu preço de negociação no mercado. Esta variação pode ser influenciada pelos seguintes factores: (i) a situação da empresa, que pode ser avaliada através da análise dos resultados trimestrais e anuais apresentados, nível de endividamento, investimentos realizados e potencial de crescimento; (ii) eventos societários como a distribuição de dividendos, aumentos de capital e ofertas públicas de aquisição; (iii) o comportamento do sector e a conjuntura económica a nível local, regional e global; e (iv) a evolução dos mercados financeiros, pois, os mercados de valores mobiliários são mercados livres e abertos à participação de investidores nacionais e estrangeiros; assim, são permanentemente influenciados pelo comportamento dos restantes mercados.

As *obrigações* são valores mobiliários com uma duração limitada que representam uma parte de um empréstimo contraído por uma empresa ou entidade junto dos investidores. Ter obrigações significa, portanto, ser credor de um emitente. Depois de um determinado período, o investidor terá direito a receber o valor que inicialmente investiu e periodicamente receberá juros, se estes tiverem sido acordados.

As *unidades de participação* são as parcelas em que se divide o património de um fundo de investimento. A duração das unidades de participação deve ser equivalente ao prazo de duração do fundo. Os fundos de investimento são organismos de investimento colectivo constituídos pelas poupanças de vários investidores. O conjunto dessas poupanças constitui um património dividido em partes iguais, com as mesmas características e sem valor nominal.

Os *títulos de participação* são valores mobiliários tendencialmente perpétuos que conferem o direito a uma remuneração com duas componentes: uma fixa e outra variável. Tanto a remuneração fixa como a variável são determinadas sobre uma percentagem do valor nominal do título de participação. Os títulos de participação podem ser emitidos por empresas públicas e por sociedades anónimas pertencentes maioritariamente ao Estado. Só são reembolsáveis se as entidades que os emitiram o decidirem, mas nunca antes de terem decorrido 10 anos desde a sua emissão, ou se essas entidades entrarem em falência.

Existem produtos financeiros que permitem aos investidores, não só controlar o risco e assegurar a rendibilidade do investimento, garantindo o preço de um bem no futuro,

como também tirar partido de uma previsão sobre a evolução dos preços. Este tipo de produtos designam-se: *contratos de futuros*. São, portanto, contratos padronizados de compra e venda a prazo, pelo qual duas partes (o comprador e o vendedor) acordam um preço relativo a uma transação futura de determinado produto ou ativo.

Os *contratos de opções* são também contratos estabelecidos entre duas partes pelos quais o comprador adquire o direito de comprar ou vender, durante um certo período de tempo, um ativo por um preço estabelecido no momento de celebração do contrato, pagando para isso um prémio.

1.2. Apresentação da Finantech

Tradicionalmente, os mercados de valores mobiliários funcionavam em grandes salas onde os corretores transmitiam com o auxílio de gestos e sinais, as ordens dos seus clientes (CMVM, 2011). A entidade que geria o mercado registava as diversas ordens de compra e venda, calculava a cotação de cada valor mobiliário e dava como realizadas as operações, promovendo a sua liquidação.

Atualmente a negociação realiza-se através de sistemas informáticos. Os corretores trabalham a partir dos seus escritórios, onde introduzem as ordens dos clientes no sistema informático de negociação do mercado, passando aquelas a designarem-se por ofertas. Quando se encontram duas ofertas de sentido inverso, o sistema realiza a operação de compra e venda e informa os corretores que as enviaram que as ofertas foram executadas. As ordens de bolsa podem ser dadas por escrito, oralmente ou através da Internet. Devem conter sempre os seguintes elementos: a identidade do investidor que dá a ordem, o tipo de ordem (compra ou venda), o valor mobiliário que se pretende negociar, a quantidade e o preço.

A Finantech é fundada em Setembro de 1994, quando um grupo de antigos colaboradores da Bolsa de Valores do Porto detecta várias lacunas nos sistemas intermediários de informação financeira que operavam no mercado de capitais português. O início da atividade ficou marcado pelo desenvolvimento de aplicações para a Corretora Atlântico e para o Banco Comercial de Macau (posteriormente integrados no Banco Comercial Português, atualmente Millenniumbcp).

A Finantech é a empresa que lidera em Portugal o desenvolvimento de sistemas de informação para o mercado de capitais. Os seus clientes são sobretudo instituições financeiras: bancos comerciais e de investimento, corretoras, salas de negociação e bolsas de valores.

Numa área de negócio em constante crescimento e mutação, as soluções são permanentemente aperfeiçoadas e adaptadas aos requisitos de cada cliente. É ainda

garantido um sistema de manutenção e monitorização permanente e fornecido um serviço de apoio ao cliente para a assistência na resolução de problemas.

A Finantech conta com importantes parcerias tecnológicas e de negócio, nomeadamente, com a *Microsoft*, *Oracle*, *Cisco* e *Tenfore Systems*, que permitem fortalecer em termos de fiabilidade e performance as soluções fornecidas.

O principal produto que a Finantech disponibiliza aos seus clientes é a *Plataforma Sifox*. Este conjunto de aplicações representa uma solução completa de *BackOffice*, *MiddleOffice* e *FrontOffice*. Ou seja, todo o processo é acompanhado, desde a intenção de compra de um investidor até à conclusão da operação, com validação de posições e escolha do sistema de encaminhamento.

O *Sifox BackOffice* consiste numa aplicação integrada e multidivisa, que suporta na globalidade as necessidades de processamento de bancos e corretores, nas operações de um vasto conjunto de instrumentos financeiros. Integra módulos de processamento e gestão de clientes, ordens e preços de valores mobiliários, negociação de títulos e gestão de carteiras e de contas financeiras; assegurando constantemente o cumprimento de todas as obrigações legais, nomeadamente o envio de relatórios (diários) às entidades de supervisão e aos clientes.

O *Sifox MiddleOffice* integra todas as funcionalidades de negociação, que incluem a gestão e encaminhamento de ordens de clientes, controlo de saldos, consulta de dados de mercado e integração em tempo real com o ambiente de *BackOffice*. A base de dados que suporta o *MiddleOffice* representa o repositório das entidades e atributos necessários às funcionalidades de negociação, nomeadamente, clientes, títulos, praças e ordens.

O *Sifox FrontOffice* é responsável pela recepção dos dados de mercado e notícias económicas, desenvolvimento de terminais de negociação, terminais de informação e negociação na Internet.

1.3. Apresentação do ServerDealWeb

O SDW está integrado na plataforma Sifox, na área de *FrontOffice (FO).* A partir da Figura 1.1 é possível enquadrar o SDW no contexto da empresa.

O acesso à informação pública (ou *market data*) dos mercados financeiros pode ser feito através das bolsas de valores (exemplos: *Euronext, BME* e *Nasdaq*) ou de fornecedores de informação (exemplos: *Reuters*, *MorningStar* e *Sungard*).

A ligação dos mercados financeiros à plataforma Sifox é feita através do *FeedHandler*.

O *FeedHandler* destina-se exclusivamente ao tratamento, processamento e distribuição de dados de mercado, em tempo real. A grande volatilidade e liquidez dos mercados atuais, com cada vez mais quantidade de informação a ser distribuída num

tempo cada vez mais curto, impôs a criação deste serviço. A informação é disponibilizada de forma normalizada e consistente a outros componentes do FO, nomeadamente, ao *ServerDeal*.

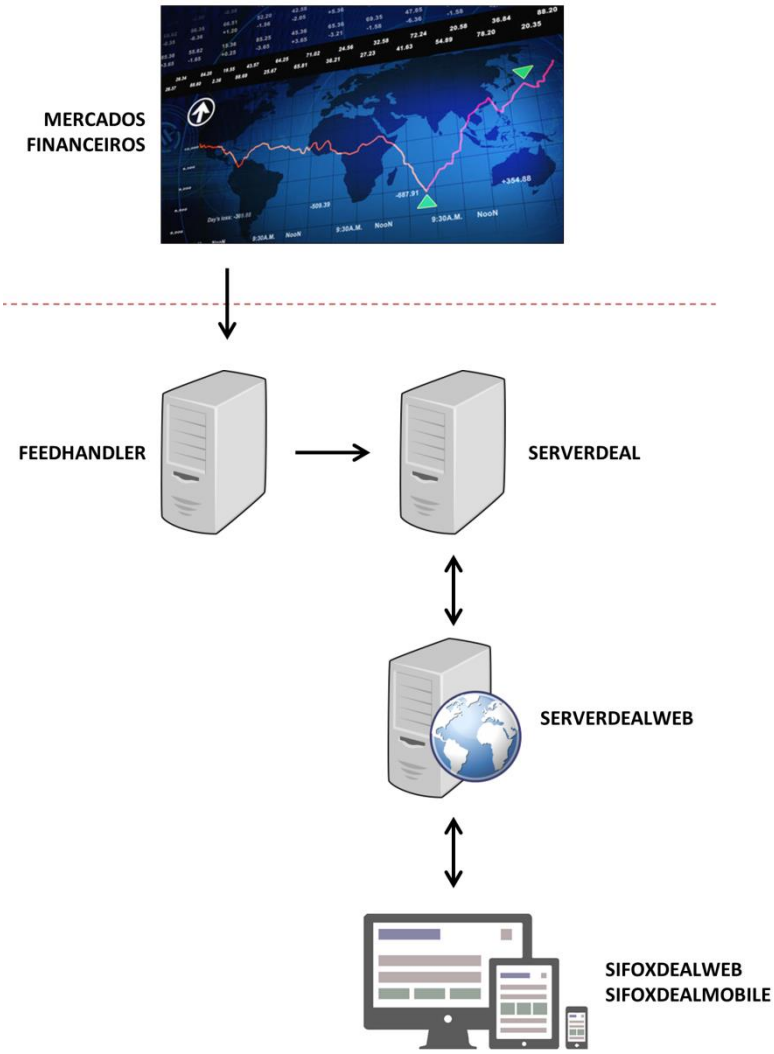


Figura 1.1: Enquadramento do *ServerDealWeb* na plataforma Sifox

O *ServerDeal* é o serviço que fornece informação financeira ao *SDW* e executa todos os seus pedidos, configurações e atualizações. O *ServerDeal* é o ponto de acesso comum a todos os intermediários e investidores.

O SDW é um servidor cujo principal objetivo é fornecer dados de mercado em tempo real a aplicações Web, nomeadamente, o *SifoxDealWeb*, *SifoxDealMobile* e páginas Web. Destina-se não só a clientes institucionais mas também a investidores particulares, com acesso a todas as funcionalidades normalmente disponíveis nos terminais profissionais. Os intermediários ou investidores que se ligam ao SDW utilizam as seguintes plataformas: *SmartClient Windows* (*.NET*); *iOS* (*Objective-C*);

Android (*Java*) e os principais navegadores disponíveis — *Internet Explorer, Google Chrome, Firefox e Safari* (HTML/ JavaScript).

As principais funcionalidades do SDW são:

- *Visualização de dados de mercado* - títulos, praças e cotações; pesquisa de títulos; profundidade; *ticker* de negócios e inserção de filtros; gráficos *intraday* e de histórico de um título; janela de carteira;
- *Alertas* - sempre que uma determinada condição sobre um título de mercado é ultrapassada (por exemplo: se a variação de uma cotação for superior a X ou se baixar até ao valor Y), o utilizador é avisado através de uma interface gráfica;
- *Negociação rápida* - possibilidade de introduzir ordens de compra ou venda, definindo o título, a quantidade, o preço que se deseja que a ordem seja inserida e validade;
- *Livro de ordens* - visualizar todas as ordens que foram inseridas e os seus parâmetros, qualquer que seja o seu estado; para além disso permite ainda cancelar, modificar e confirmar ordens;
- *Ticker de ordens* - esta funcionalidade permite ter acesso a uma janela de registo do que vai acontecendo a todas as ordens inseridas;
- *Saldos* - visualização do saldo de negociação, incluindo os ativos elegíveis (que inclui ordens de compra que tenha no sistema), a compra máxima de ativos elegíveis (que diz o máximo que a sua alavancagem lhe permite utilizar), os capitais próprios e o grau de alavancagem;
- *Gestão de acessos e autenticação de clientes* - atribuindo a cada nova ligação as respectivas permissões e configurações.

O SDW é uma DLL (biblioteca de vínculo dinâmico), desenvolvida em C++, invocada a partir de um servidor Web (incluído com o *Windows*), o *Internet Information Services* (IIS). Uma DLL é um arquivo executável que atua como uma biblioteca de funções. O vínculo dinâmico possibilita que um processo execute uma função que não é parte do seu código executável.

De forma a garantir uma elevada taxa de disponibilidade e escalabilidade, a Finantech define geralmente um conjunto fixo de instâncias de IIS para cada cliente, de acordo com um conjunto de indicadores, nomeadamente, número máximo de utilizadores em simultâneo, quantidade de tráfego e ordens transacionadas diariamente, entre outros factores.

Existe um mecanismo definido de balanceamento de carga responsável por gerir a distribuição das ligações. Na Figura 1.2 verifica-se que, para cada utilizador, existem pelo menos duas ligações: uma permanente, para a recepção contínua de dados de

mercado (do SDW para o cliente); e pelo menos mais uma ligação, para comunicar com o servidor, sempre que é necessário trocar informações entre ambos ou alterar alguma subscrição.

Todas as ligações são estabelecidas por afinidade, ou seja, depois da autenticação num dos SDW é garantido que os pedidos subsequentes são dirigidos para o mesmo SDW. Depois de uma subscrição bem-sucedida, a informação sobre um determinado item flui automaticamente sempre que ocorre alguma atualização do mesmo, não sendo necessário qualquer intervenção por parte do utilizador final.

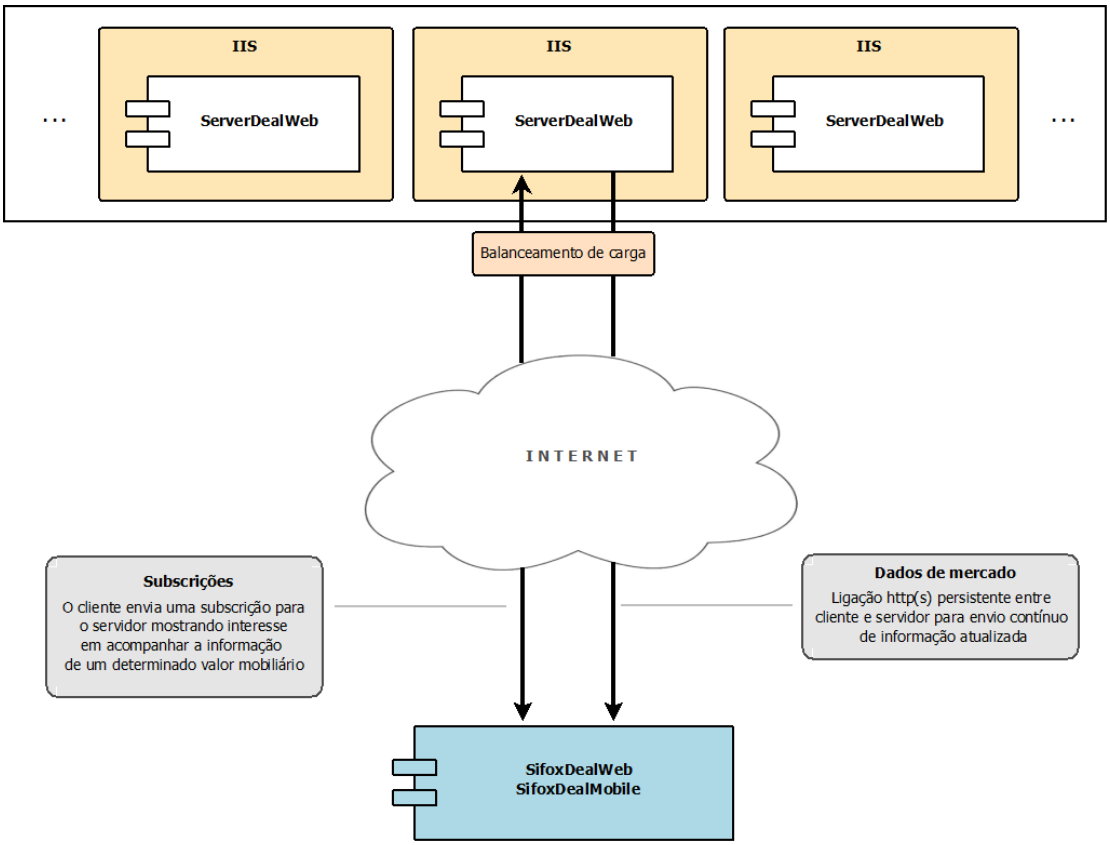


Figura 1.2: Ligação entre SDW e *SifoxDealWeb* e *SifoxDealMobile*

Este modelo implica que a cada pedido do cliente (por exemplo, envio de uma nova ordem, pedido de dados mercado, livro de ordens ou carteira de títulos), seja enviado um pedido HTTP, que pode implicar a abertura de uma nova ligação.

Relativamente às interfaces internas, representadas na Figura 1.3, existem ligações a três serviços que fornecem dados ao SDW: (i) *ServerDeal* — dados de mercado, ou seja, cotações, ordens e praças; (ii) *Risco* — um serviço de confirmação de ordens e classificação de clientes; e (iii) *Alertas* — para avisar o cliente se determinados parâmetros são ultrapassados. O SDW está ainda ligado a duas bases de dados: SAA (autenticação de utilizadores) e SMS (dados de clientes, grupos e contas).

Para transportar mensagens entre o SDW e o cliente, é utilizado o protocolo HTTP. No caso dos clientes com navegadores Web recebem em formato de texto que é convertido numa estrutura de dados. Para todos os outros clientes, o HTTP é usado para enviar mensagens baseadas no protocolo FIX (*Financial Information Exchange Protocol*).

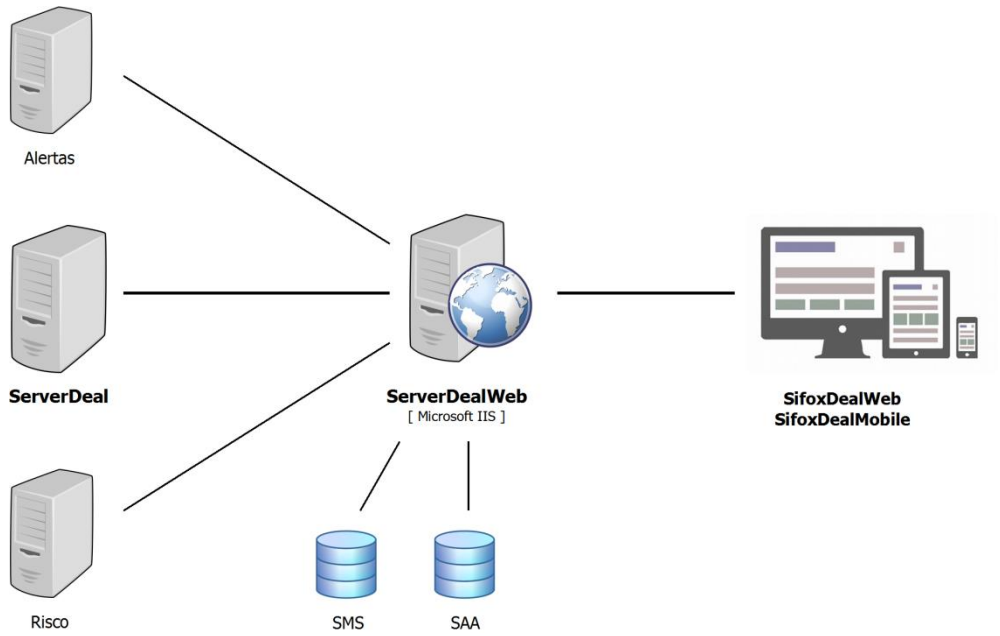


Figura 1.3: Ligações do SDW

1.4. Definição de objectivos

O processo de unificação dos mercados financeiros e a homogeneização da economia mundial que se verificou no final do século XX, vieram aumentar exponencialmente a presença de tecnologias de informação no sector financeiro (Silva, 2006). Atualmente, a complexidade, sofisticação e grau de exigência das instituições e dos mercados financeiros constituem um enorme desafio para os sistemas informáticos: diversidade de instrumentos financeiros, pluralidade de canais de atendimento e acesso, número elevado de transações bancárias efetuadas diariamente, monitorização rigorosa dos investimentos, automatização de operações financeiras (nomeadamente, débitos diretos e pagamentos de serviços), vasto conjunto de requisitos de segurança, entre muitos outros factores. Para além disso, a necessidade de mobilidade dos clientes finais e de garantir o acesso aos mercados financeiros através das plataformas móveis, exige o desenvolvimento de aplicações Web robustas, rápidas e seguras.

O objectivo deste trabalho é desenvolver um protótipo de um sistema que permita diminuir a latência na comunicação e aumentar a coerência de dados entre o *ServerDeal* e as aplicações cliente *SifoxDealWeb* e *SifoxDealMobile*.

Tem-se verificado um aumento gradual de utilizadores destas duas aplicações em todos os clientes da Finantech. De forma a evitar a sobrecarga do sistema e, consequentemente, a quebra de ligações ou atraso nas comunicações, a arquitetura atual tem de ser revista. Para além disso, as exigências dos intermediários e investidores e a competitividade do mercado também são cada vez maiores.

Latência é o tempo que demora um determinado evento que ocorreu num dos lados da ligação (no cliente ou servidor), a ser registado do outro lado (Bozdag, Mesbah, & Deursen, 2009). Por exemplo, num sistema de mensagens entre dois utilizadores, latência é o tempo que demora entre a mensagem ser enviada (quando o utilizador clica em "Enviar") e o momento em que aparece na janela do outro utilizador.

Coerência de dados significa que os dados do cliente e do servidor encontram-se no mesmo estado. Ou seja, quando ocorre uma alteração num dos lados, o outro lado é imediatamente atualizado.

Alguns exemplos onde a latência e a coerência de dados são importantes: páginas Web de leilões, cobertura de eventos desportivos, bancos ou corretoras, sistemas de mensagens instantâneas, aplicações com sugestões que se completam automaticamente, criação e manipulação de gráficos ou diagramas em tempo real, entre outros.

Com este trabalho pretende-se ainda: (i) tornar o SDW um serviço independente do IIS, dado que isto obriga uma adaptação constante do SDW ao IIS, sempre que este é atualizado pela *Microsoft*; (ii) uniformização de implementações, uma vez que atualmente existem diferentes implementações para clientes que utilizam o navegador para aceder ao SDW e para todos os outros clientes; (iii) simplificação da arquitetura geral do SDW, eliminando a necessidade de manter duas ligações para cada utilizador; e (iv) otimização de todo o fluxo e gestão de informação, para além dos sistemas de análise de risco ou de execução automática.

A solução para atingir os objetivos definidos passa por encontrar um protocolo de comunicação que permita estabelecer ligações na Web de forma eficiente. Ou seja, um protocolo que permita comunicações assíncronas e bidirecionais entre aplicações e serviços Web, utilização eficiente da largura de banda e cuja implementação esteja disponível para a maioria das plataformas tecnológicas dos atuais clientes do SDW.

Ao reduzir a latência e aumentar a coerência de dados na comunicação entre o *ServerDeal* e as aplicações *SifoxDealWeb* e *SifoxDealMobile*, será possível otimizar a utilização de recursos computacionais, oferecer mais conteúdos e funcionalidades aos clientes e aumentar a qualidade geral do serviço prestado.

1.5. Evolução do trabalho

A partir da Figura 1.4 é possível identificar as principais etapas de desenvolvimento deste trabalho de estágio. Os dias iniciais foram de apresentação da Finantech e de conhecimento da sua história, organização interna e principais produtos que disponibiliza aos clientes.

O primeiro mês de trabalho foi dedicado ao estudo do HTML5 e das suas novas funcionalidades, do protocolo *websocket* (em particular, *o pedido inicial de ligação* e o formato das mensagens) e da *Application Programming Interface* (API) *websocket*, que permite às aplicações desenvolvidas controlar o protocolo e responder a eventos despoletados pelo servidor.

Seguiu-se a implementação de um servidor de eco baseado em *websocket*. Este servidor aceita a ligação de apenas um cliente, recebe deste uma mensagem e retorna a mesma mensagem para o cliente (fazendo *eco* desta). A etapa seguinte começou por adaptar o anterior servidor de forma a ser capaz de aceitar vários clientes em simultâneo. Sempre que algum cliente se liga ao servidor é enviada uma mensagem igual para todos os outros clientes que estão ligados.

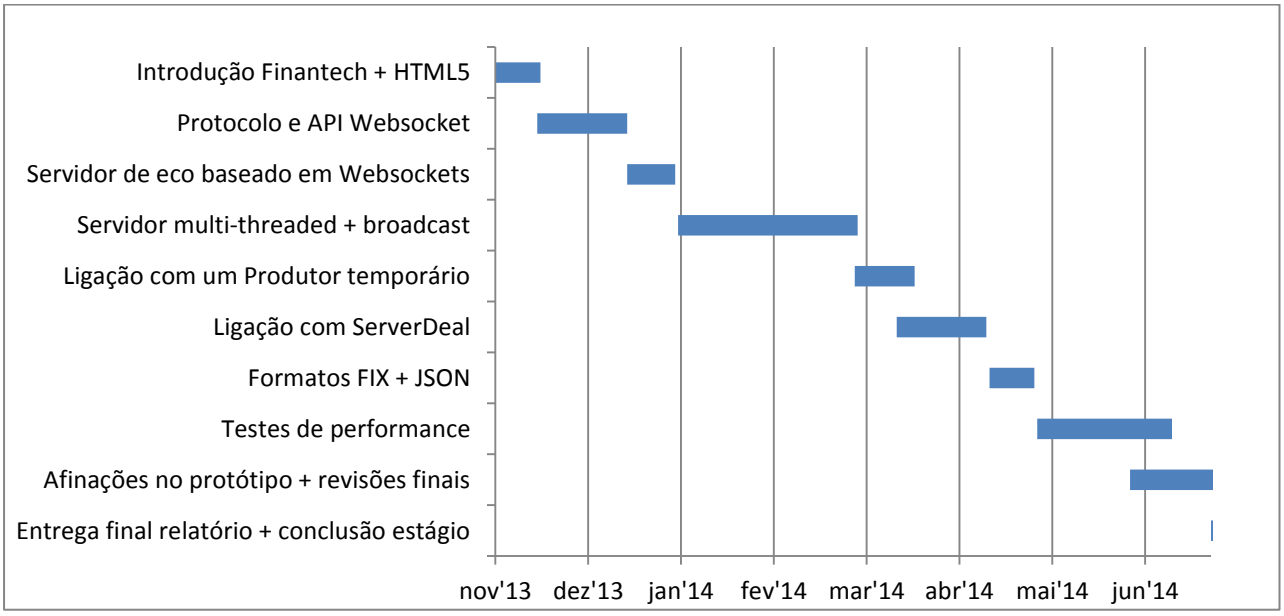


Figura 1.4: Etapas de desenvolvimento do trabalho

No início de Março é criado um servidor auxiliar (“produtor”) que se liga ao servidor principal e lhe envia continuamente mensagens; todos os clientes (“consumidores”) que estão ligados recebem a mesma informação. Depois deste período com mensagens simuladas foi estabelecida uma ligação ao *ServerDeal*. A partir de meados de Abril, o formato FIX passou a ser utilizado na troca de mensagens entre o servidor

principal e o *ServerDeal* e o formato JSON entre o servidor e o cliente de um navegador Web.

Finalmente seguiu-se uma fase de testes de performance, melhorias no protótipo e revisões finais quer do código, quer do relatório.

Ao longo do estágio foi também aprofundado o conhecimento das linguagens de programação C++11 e *JavaScript*, do protocolo FIX e do formato de mensagens JSON. Enquanto plataforma principal de desenvolvimento foi utilizado o *Microsoft Visual Studio 2013*.

Depois da introdução aos mercados financeiros e à Finantech, e da apresentação do ServerDealWeb, este relatório apresenta, no segundo capítulo, várias alternativas tecnológicas para estabelecer um canal de comunicação bidirecional entre um cliente e um servidor na Web. No terceiro capítulo é apresentado o protocolo e a API *websocket* e algumas recomendações práticas para reforçar a segurança de serviços e aplicações Web que utilizam *websocket*. No quarto capítulo é apresentado o protótipo que foi desenvolvido, nomeadamente, a escolha da tecnologia e a descrição da arquitetura do *software.* No último capítulo são exibidos os resultados de um conjunto de testes de performance ao protótipo desenvolvido e alguns testes comparativos entre o protótipo e o atual SDW.

2. Tecnologias relacionadas

Neste capítulo são apresentadas as principais alternativas tecnológicas de comunicação para fornecer dados a aplicações Web em tempo real, que suportam comunicação assíncrona de eventos entre cliente e servidor.

2.1. Introdução

O protocolo HTTP (*Hypertext Transfer Protocol*) é o protocolo de comunicação mais utilizado na Internet entre sistemas distribuídos, para enviar e receber informações. É um protocolo da camada aplicacional (do modelo OSI), que não guarda o estado da interação entre cliente e servidor (Carreira, 2008).

O modelo cliente-servidor oferece algumas vantagens, nomeadamente: informação centralizada; facilidade de manutenção; suporte a múltiplos clientes em simultâneo; facilidade de gestão e segurança da informação; agrupamento de serviços e partilha de recursos. Porém, existem também desvantagens, como sejam: em caso de falha do servidor, todos os clientes deixam de poder realizar pedidos; se a segurança do servidor for quebrada, o atacante tem acesso a dados dos utilizadores registados; o servidor pode não suportar uma grande quantidade de pedidos simultâneos, estando sempre limitados a um número fixo (e máximo) de ligações.

Para estabelecer uma comunicação HTTP entre um cliente (por exemplo, um navegador Web) e um servidor é necessário estabelecer, previamente, uma ligação TCP. A partir da Figura 2.1 é possível identificar todas as trocas de mensagens que ocorrem neste processo.

Numa ligação TCP o servidor abre um *socket* e espera passivamente por ligações. No outro extremo, o cliente, inicia a ligação enviando um pacote TCP com a *flag* SYN ativa. Espera-se que o servidor aceite a ligação enviando um pacote SYN+ACK. Se durante um determinado período de tempo esse pacote não for recebido ocorre um *timeout* e o pacote SYN é reenviado. O estabelecimento da ligação é concluído por parte do cliente, confirmando a aceitação do servidor respondendo com o envio de um pacote ACK.

Um *socket* é um mecanismo de comunicação necessário para a transmissão de dados através de protocolos de transporte (exemplo: TCP e UDP). Permite que duas ou mais aplicações troquem informações, seja no mesmo computador ou em computadores diferentes. Um *socket* é a combinação entre o endereço IP e a porta de um determinado processo a ser executado nesse endereço IP.

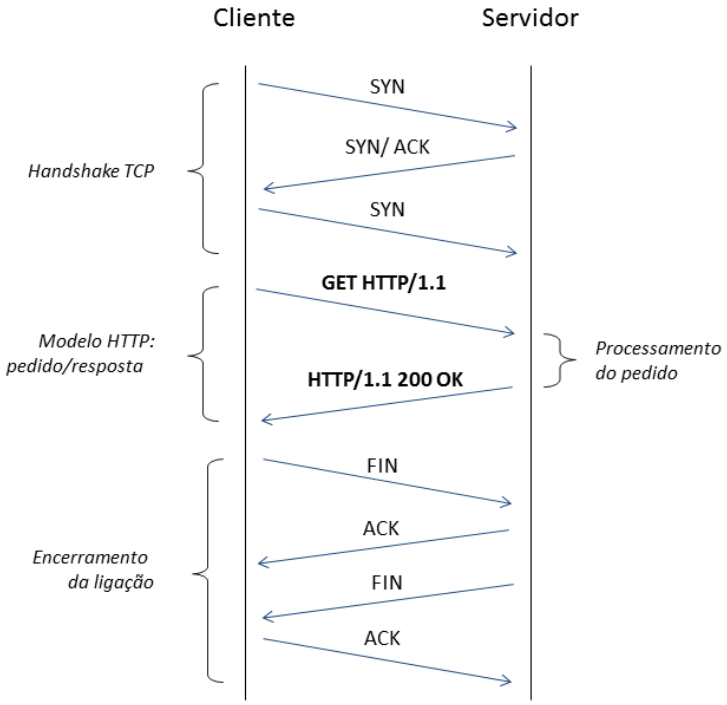


Figura 2.1: Sessão HTTP

Depois de aberta a ligação, o cliente envia um pedido HTTP para o servidor, através de um endereço URL (*Uniform Resource Locator*), e aguarda por uma resposta. Na Figura 2.2 é apresentada a estrutura de um endereço URL (Kumar, 2014). No pedido está indicado o seu método (habitualmente *GET* ou *POST*), a identificação do recurso pretendido e os parâmetros do pedido.

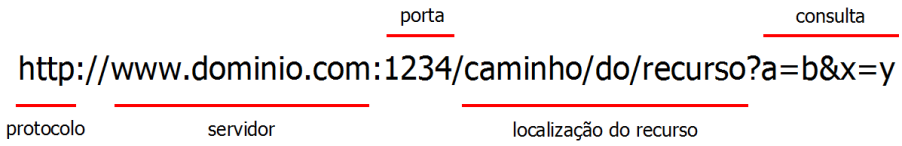


Figura 2.2: Formato de um endereço URL

Do outro lado, o servidor, que armazena conteúdos e fornece recursos, processa o pedido e envia a resposta para o cliente, que contém informações sobre o estado de conclusão do pedido (exemplos: 200 = OK, 404 = Recurso inválido, não existe no servidor; 500 = Erro interno do servidor). Em caso de sucesso, contém o conteúdo solicitado pelo cliente, no corpo da mensagem.

Para terminar a ligação, ocorre um processo em quatro fases. Primeiro é enviado um pacote com a *flag* FIN ativa, ao qual deverá receber uma resposta ACK. Por sua vez, o outro interlocutor irá proceder da mesma forma, enviando um FIN ao qual deverá ser respondido um ACK.

O protocolo HTTP é *half-duplex*, o que significa que os dados podem ser transmitidos nas duas direções (cliente-servidor) alternadamente. Sempre que o cliente pretender aceder a informações atualizadas tem de efetuar um novo pedido para o servidor e aguardar pela resposta. Este tempo de espera pode ser prejudicial em várias situações, nomeadamente, no acesso a preços de valores mobiliários, notícias, venda de bilhetes, padrões e comportamento de tráfego, redes sociais e leituras de dispositivos médicos (Lubbers & Greco, 2012).

Nos últimos anos tem-se verificado uma alteração no padrão de utilização da Internet. A antiga rede de páginas HTML estáticas está a dar lugar a um conjunto de plataformas complexas que permitem a execução de sistemas distribuídos interativos, com atualização em tempo real, personalizados e com elevadas capacidades de processamento e armazenamento. Ao considerar este novo paradigma, o modelo de comunicação pedido/ resposta do protocolo HTTP é desadequado para atender aos novos requisitos (Varela, 2011). É neste contexto que surgem algumas tentativas para implementar um modelo de comunicação bidirecional, que permite a comunicação entre cliente e servidor nos dois sentidos.

> *Simulating bi-directional browser communication over HTTP is error-prone and complex and all that complexity does not scale. Even though your end users might be enjoying something that looks like a real-time web application, this "real-time" experience has a high price tag. It's a price that you will pay in additional latency, unnecessary network traffic and a drag on CPU performance.* (Lubbers P. A., 2010)

2.2. Polling

Uma das alternativas para fornecer às aplicações Web dados em tempo real é utilizar a técnica de *polling*, que repete regularmente o mesmo pedido HTTP ao servidor. Existem dois tipos: *short polling* e *long polling.*

2.2.1. Short polling

Short polling é implementado fazendo um pedido ao servidor, em intervalos regulares de tempo (poucos segundos). Se os dados foram alterados comparativamente aos que estão no cliente, o servidor envia novamente os dados atualizados; caso contrário, o servidor responde com uma mensagem em branco. A Figura 2.3 ilustra o funcionamento do *short polling* ao longo do tempo (Swamy & Mahadevan, 2011). Verifica-se uma sequência de pedidos por parte do cliente ao servidor, a cada *n* segundos.

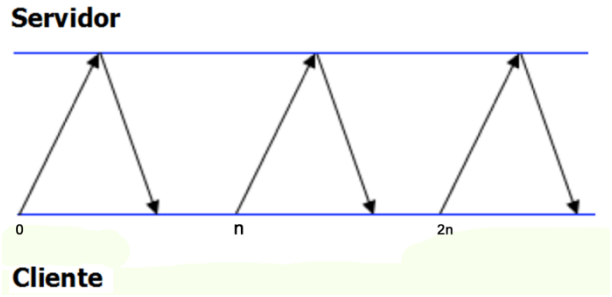


Figura 2.3: *Short polling*

Esta técnica é eficiente apenas quando o cliente conhece o intervalo exato de tempo que os dados são atualizados no servidor. Caso contrário, sobrecarrega o servidor com pedidos desnecessários, o que implica a abertura e fecho de muitas ligações. Existe ainda uma sobrecarga no uso do processador do servidor ao verificar constantemente se existem novos dados (Jõhvik, 2011).

2.2.2. Long polling

Long polling permite que a ligação entre o cliente e o servidor se mantenha aberta por um determinado período de tempo, até que seja enviada uma resposta do servidor (Swamy & Mahadevan, 2011). Inicialmente, o cliente envia o seu pedido para o servidor e aguarda por uma resposta; se os dados no servidor forem alterados dentro do prazo definido é enviada uma resposta para o cliente contendo os dados atualizados e a ligação é fechada; caso contrário, o servidor envia uma resposta para terminar o pedido que estava pendente ou o cliente cria um novo pedido.

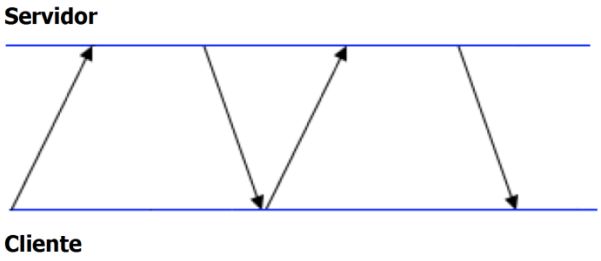


Figura 2.4: *Long polling*

Apesar da simplicidade e facilidade de configuração, a partir da Figura 2.4 verifica-se que o *long polling* é desadequado para os casos onde exista uma elevada frequência de atualizações. Este cenário gera uma enorme quantidade de mensagens entre o cliente e o servidor e obriga o servidor à gestão de uma grande quantidade de ligações abertas em simultâneo.

2.3. BOSH

O protocolo BOSH (*Bidirectional-streams Over Synchronous HTTP*)[1], baseado na técnica de *long polling*, permite simular um canal de mensagens bidirecional sobre o protocolo HTTP mantendo abertos múltiplos pares pedido/ resposta entre cliente e servidor (Laine & Säilä, 2012).

O protocolo BOSH foi inicialmente criado com o objetivo de transportar o protocolo XMPP (*Extensible Messaging and Presence Protocol*) através do HTTP. O XMPP é um protocolo aberto, baseado em XML, para sistemas de mensagens instantâneas. Os pacotes XML usados no BOSH são iguais aos pacotes de XMPP quando este é executado sobre TCP. Ou seja, BOSH é um mecanismo que permite a troca do protocolo XMPP entre um navegador Web e um servidor BOSH sobre HTTP, tal como está ilustrado na Figura 2.5. De tal forma que um cliente Web XMPP poderá utilizar um navegador e utilizar BOSH para comunicar com um servidor XMPP.

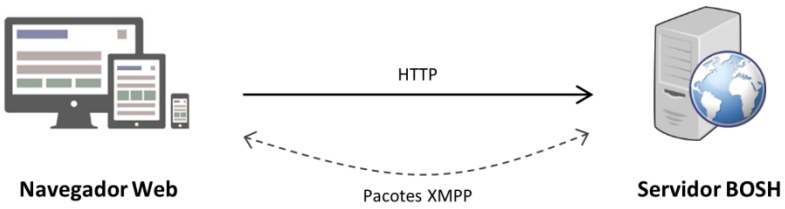


Figura 2.5: Funcionamento do protocolo BOSH

2.4. Streaming

Ao contrário do modelo habitual cliente-servidor, onde o servidor termina a ligação com o cliente depois de processar o seu pedido, no modelo *HTTP Streaming* ou *Comet*, o servidor mantém uma ligação aberta e persistente com o cliente, mesmo depois de enviar uma resposta ao cliente.

2.4.1. Server-Sent Events

Server Sent Events (SSE)[2] normalizam as técnicas de *streaming* e enquadram-se no modelo de comunicação orientado a eventos. Neste tipo de sistemas um cliente começa por se registar no servidor, indicando as classes de eventos ou notificações

[1] http://xmpp.org/extensions/xep-0124.html - XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH); *"This specification defines a transport protocol that emulates the semantics of a long-lived, bidirectional TCP connection between two entities (such as a client and a server) by efficiently using multiple synchronous HTTP request/response pairs without requiring the use of frequent polling or chunked responses."*

[2] http://dev.w3.org/html5/eventsource/ - "*This specification defines an API for opening an HTTP connection for receiving push notifications from a server in the form of DOM events. The API is designed such that it can be extended to work with other push notification schemes such as Push SMS.*"

que pretende receber. Quando o servidor detecta a ocorrência de algum evento envia a respectiva notificação para aqueles que o subscreveram.

Os SSE têm como objectivo tornar nativa a possibilidade de atualizações em tempo real. A API SSE está contida na interface *EventSource* (Cravens, 2013). Para abrir uma ligação com o servidor de forma a receber eventos é necessário criar um objeto *EventSource*, indicando o URI do *script* que produz os eventos.

O cliente envia um único pedido e o servidor mantém a ligação aberta indefinidamente (ou por um período tempo), enviando continuamente atualizações dos dados. A partir da Figura 2.6 verifica-se que durante a utilização de SSE é criado um canal unidirecional e persistente entre o cliente e o servidor. Desta forma, o servidor é capaz de enviar dados para o cliente sempre que necessário, sem qualquer pedido explícito por parte deste.

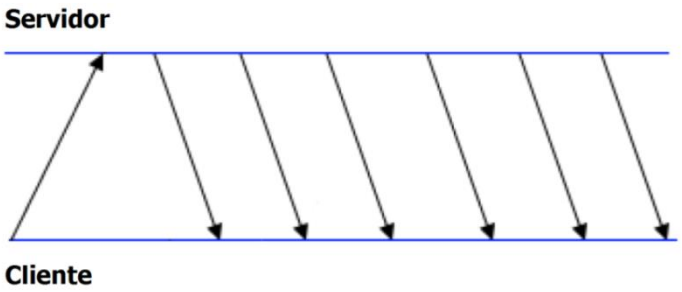


Figura 2.6: HTTP Streaming

As duas maiores diferenças entre o SSE e o *long polling* são o facto de os SSE serem geridos diretamente pelo servidor e o cliente ter apenas de aguardar, passivamente, pela chegada de novas mensagens.

Esta técnica apresenta duas vantagens que realçam a sua simplicidade na criação, análise e manipulação de eventos: é baseada em HTTP normal e em texto simples (Vinoski, 2012). Porém, também existem algumas desvantagens: (i) se o cliente precisar de comunicar com o servidor, por exemplo, para alterar alguma configuração ou subscrição, será obrigado a criar uma nova ligação com o servidor; (ii) nem todos os navegadores, em particular, o *Internet Explorer*; têm suporte nativo para esta tecnologia; (iii) por vezes os servidores *proxy* não transferem *bytes* se a ligação HTTP não estiver fechada, o que torna o *streaming* inapropriado; e (iv) do lado do servidor existe a alocação de um grande número de recursos para manter várias ligações abertas em simultâneo.

2.4.2. Forever frame

A técnica de *forever frame* utiliza um *iframe* escondido, embebido no cliente que se irá ligar ao servidor que pretende receber informação. Quando o servidor recebe a ligação

iframe, começa a enviar os dados pedidos, rodeados pelas etiquetas <*script*></*script*>, que contém uma função *JavaScript* para ser chamada do lado do cliente. Os dados que o cliente envia são os argumentos desta função. Em seguida, a informação contida entre as etiquetas <*script*></*script*> são executadas pelo cliente por uma determinada ordem.

Contudo, esta solução apresenta alguns inconvenientes: (i) os códigos de ambos os lados (cliente e servidor) têm de ser conhecidos e estar diretamente relacionados, devido ao facto do servidor ter de saber qual a função em *JavaScript* do cliente que irá despoletar o envio dos dados; (ii) sempre que a ligação é perdida (por exemplo, devido a *timeouts*), o *iframe* não tem capacidade de a recuperar; (iii) não há confirmação por parte do cliente que confirme a recepção dos dados; (iv) esta técnica envia dados incrementalmente, mas o cliente armazena toda a informação recebida, o que constitui um problema de gestão de memória; e (v) servidores *proxy*, *routers* e outros dispositivos de redes podem interromper o fluxo de dados, impedindo o cliente de aceder a algumas atualizações.

2.5. XMLHttpRequest

Uma das tecnologias de programação Web que permitiu uma grande evolução no desenvolvimento de aplicações interativas, a partir de 2005, foi o AJAX (acrónimo de *Asynchronous JavaScript and XML*). Esta técnica suporta a troca de pequenas quantidades de dados entre cliente e servidor, assincronamente (ZEPEDA & CHAPA, 2007), e disponibiliza ao cliente métodos como *open, send* e *responseText*.

XMLHttpRequest (XHR) é uma API disponível em linguagens de *script* para navegadores Web. XHR pode ser considerado um objeto *JavaScript* que torna possível a comunicação assíncrona com o servidor, sem a necessidade de recarregar a página por completo. A Figura 2.7 representa o processamento de um pedido AJAX.

Esta especificação define um conjunto de métodos e parâmetros que permitem a ligação ao servidor, a transferência de dados e a possibilidade de aguardar por uma resposta, evitando recarregar toda a página da qual foi enviado o pedido (Smullen & Smullen, 2008). A resposta do servidor pode ser recebida através de JSON, XML, HTML ou como texto simples. Os dados recebidos podem também ser avaliados pelo código do lado do cliente e serem usados diretamente para alterar o DOM[3] do documento ativo na janela do navegador sem carregar um novo documento de página (Garrett, 2005).

[3] O *Document Object Model* (DOM), definido pela W3C, é uma plataforma/ linguagem que permite a programas e *scripts* tenham acesso dinâmico e atualizem o conteúdo, a estrutura e o estilo de documentos. Esta especificação inclui um conjunto de APIs que permite aos programadores manipularem o DOM a partir de um *script* que esteja em execução no contexto de um navegador.

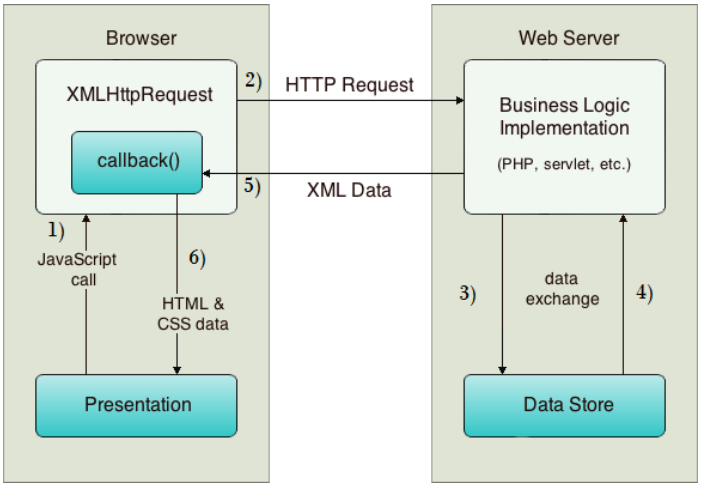


Figura 2.7: Pedido AJAX

O AJAX aumentou a capacidade de resposta das páginas Web tradicionais e permitiu aos programadores desenvolver novas funcionalidades para as suas aplicações. Os objetos XHR são utilizados para transmissões AJAX e métodos *Comet*, dado que permitem guardar os dados e reportar imediatamente o seu estado (Fan & Wang, 2010).

2.6. Web messaging

Sistemas de *Web Messaging*[4] correspondem a uma API introduzida na nova especificação do HTML5, que permite que documentos comuniquem entre si, mesmo que se localizem em origens e domínios diferentes. Ou seja, esta é uma forma dos documentos Web, em contextos de navegadores diferentes, partilharem dados sem exporem o DOM a pedaços de código externo (eventualmente malicioso).

Os sistemas de *web messaging* incluem dois sistemas que fazem parte do HTML5: (i) *cross-document messaging*: geralmente identificado pela sintaxe da função de *window.postMessage()*; e (ii) *channel messaging*: também conhecido como *MessageChannel*.

Nestes sistemas as mensagens são definidas pela interface *MessageEvent* e contêm cinco atributos, que apenas podem ser lidos: dados, origem (por exemplo: https://domain.example:80), identificador único, referência à fonte do documento (mais concretamente, um objecto *WindowProxy*) e uma lista de objetos *MessagePort.*

Porém, estes sistemas apresentam algumas desvantagens: (i) falta de mecanismos de autorização e autenticação; (ii) os únicos sistemas preparados para receber um

[4] www.w3.org/TR/webmessaging/ - "*This specification defines two mechanisms for communicating between browsing contexts in HTML documents.*"

elevado volume de mensagens exigem uma versão comercial, com custos elevados; (iii) a introdução de otimizações e configurações manuais num sistema genérico de *web messaging*, de forma a melhorar a performance e adaptar consoante as necessidades, é pouco recomendável e bastante complexo.

2.7. Web Real-Time Communication

Web Real-Time Communication (WebRTC)[5] representa uma técnica para melhorar as capacidades de comunicação entre navegadores, sobretudo na transmissão de áudio e vídeo em tempo real. Ilustrada na Figura 2.8, WebRTC é uma tecnologia *peer-to-peer*[6] para a Web, independente dos protocolos de ligação utilizados nas redes dos clientes (Isak, 2013).

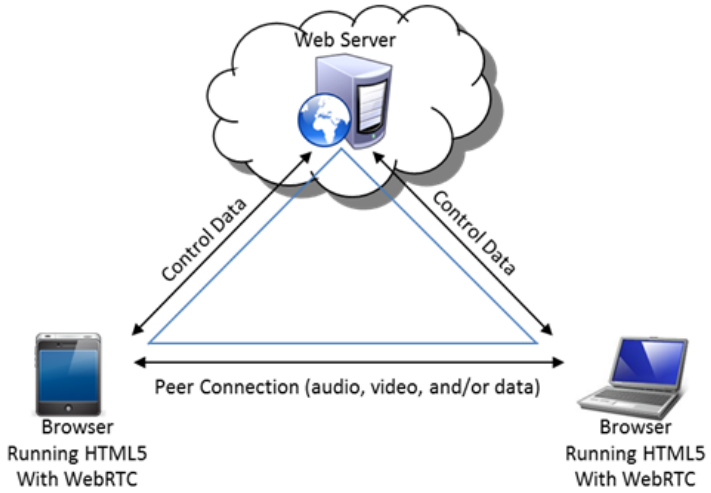


Figura 2.8: Arquitetura WebRTC

WebRTC pertence à especificação do HTML5 e inclui um conjunto de APIs que permitem aos navegadores comunicarem diretamente entre si, em tempo real, sem encaminharem todos os dados através de um servidor (Hanson, 2014). Isto permite a criação de aplicações de videoconferência, trocas de mensagens e de ficheiros, utilizando apenas um navegador (atualmente *Chrome*, *Firefox* ou *Opera*), sem recurso à instalação de software adicional (*plugins*, plataformas ou livrarias).

[5] www.w3.org/TR/webrtc/ - *WebRTC 1.0: Real-time Communication Between Browsers - This document defines a set of ECMAScript APIs in WebIDL to allow media to be sent to and received from another browser or device implementing the appropriate set of real-time protocols. This specification is being developed in conjunction with a protocol specification developed by the IETF RTCWEB group and an API specification to get access to local media devices developed by the Media Capture Task Force.*

[6] *Peer to peer* ("par a par" ou "ponto a ponto") representa um tipo de arquitetura de redes de computadores onde cada um dos pontos ou nós da rede funciona tanto como cliente como servidor, permitindo partilhas de serviços e dados sem a necessidade de um servidor central.

2.8. SPDY

SPDY (pronuncia-se *speedy*)[7] é um protocolo de rede de código aberto, da camada aplicacional, atualmente a ser desenvolvido pela *Google*. Este protocolo pretende melhorar a performance das páginas Web, através de várias técnicas (Wang, Salim, & Moskovits, 2013), nomeadamente:

Compressão — o cabeçalho de cada mensagem HTTP que seja repetido ou desnecessário, é comprimido pelo cliente e servidor; o que permite a redução de largura de banda necessária para a ligação;

Multiplexed streams — Transmissão de múltiplos sinais ou fluxos de informação em simultâneo através da mesma ligação TCP, evitando-se que recursos menos importantes bloqueiem a ligação a recursos com uma prioridade maior; redução da quantidade de ligações necessárias entre cliente e servidor;

Segurança — é adicionada uma camada de sessão sobre o SSL, garantindo um maior grau de segurança à comunicação;

Pipelining — possibilidade de enviar vários pedidos HTTP numa única ligação TCP, sem aguardar pelas respetivas respostas; e permitir que o servidor inicie uma ligação com um cliente e envie informações sempre que necessário; o que possibilita uma gestão mais eficiente da largura de banda por parte do servidor (The Chromium Projects, 2013).

O protocolo SPDY não substitui o protocolo HTTP; porém, partilham o mesmo estilo e semânticas de funcionamento e organização. A partir da Figura 2.9 é possível verificar as diferenças entre o modelo tradicional de camadas TCP/IP e o protocolo SPDY.

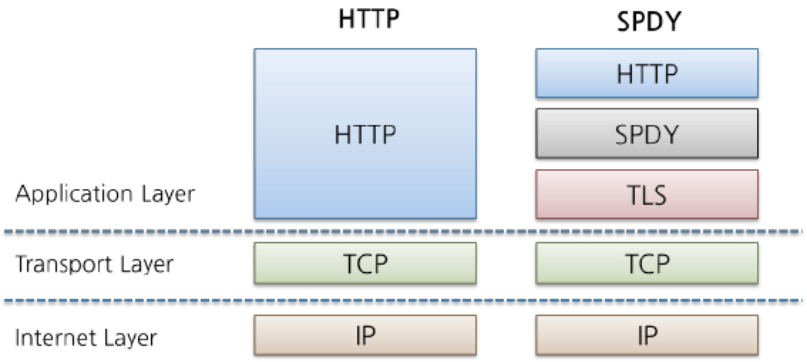


Figura 2.9: HTTP vs SPDY

[7] https://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00

O SPDY altera a forma como os pedidos e respostas são enviados através da rede, optimizando a entrega de conteúdos de aplicações e páginas Web. O que significa que todas as aplicações do lado dos servidores podem ser facilmente adaptadas para suportar o protocolo SPDY. A maioria dos navegadores atuais suporta este protocolo.
Contudo, este protocolo também apresenta algumas desvantagens, nomeadamente: (i) a necessidade de certificados SSL pode tornar as comunicações dependentes de um número limitado de organizações que emitem os certificados e colocar em risco a abertura e gratuidade da Web; (ii) a necessidade do pedido inicial SSL pode não trazer melhorias em termos de performance; (iii) o protocolo exige uma ligação por cada domínio, que é uma desvantagem no caso de existirem muitos domínios; (iv) apenas se apresenta como uma alternativa ao HTTP quando o atraso na comunicação ocorrer especificamente no HTTP; (v) o protocolo também não apresenta vantagens em situações onde o *Round Trip Time* é baixo ou quando a página Web é pequena.

2.9. PubSubHubbub

PubSubHubbub (PuSH) é um protocolo de código aberto baseado no modelo publicação/ subscrição, que permite modificar a subscrição de serviços de notícias (por exemplo, *feed RSS, Atom*), com sistemas de mensagens em tempo real. Acessível através do HTTP, o PuSH pode ser aplicado a qualquer tipo de dados (texto, imagens, som, vídeo).
O principal objetivo deste protocolo é fornecer notificações em tempo real a todos os subscritores, sem a necessidade de pedidos regulares por parte destes. Isto permite uma distribuição mais rápida de novos conteúdos, comparativamente aos serviços que existem atualmente baseados no mesmo modelo (Google Project Hosting, 2011).
O PuSH junta no mesmo pedido HTTP POST várias notificações para um único subscritor, dado que tem a capacidade de agregar múltiplos *feeds* através do elemento *atom:source.* O protocolo permite ainda a manutenção de ligações abertas e persistentes e recorre a técnicas como *pipelining* para distribuir notificações rapidamente.

Neste capítulo foi apresentado um conjunto de alternativas tecnológicas para a comunicação em tempo real, entre um cliente e um servidor, na Web: técnicas de *polling* e *streaming*, protocolos de comunicação e de rede (BOSH, SPDY, PuSH) e várias APIs (XHR, Web Messaging, WebRTC). Cada alternativa foi sucintamente descrita, destacando-se as principais vantagens e inconvenientes.
No próximo capítulo é apresentada mais uma opção: o protocolo *websocket*.

3. WebSocket

O HTML (*hypertext markup language*, ou linguagem de marcação de hipertexto) é uma linguagem de formatação de conteúdos para a comunicação na Internet, através do protocolo HTTP. O HTML separa o conteúdo da forma, ou seja, distingue o conjunto de elementos (palavras, imagens, vídeos, áudio ou documentos), das instruções que definem o formato como este conteúdo deve ser exibido. Utiliza um conjunto pré-definido de elementos para identificar diferentes tipos de conteúdo, que contêm uma ou mais etiquetas, delimitadas por sinais de menor (“<”) e maior (“>”).
Desde o começo o HTML foi criado para ser uma linguagem independente de plataformas, navegadores e outros meios de acesso. Dessa forma, evitou-se que a Internet fosse desenvolvida numa base proprietária, com formatos incompatíveis e limitados. (Tableless, 2013)

3.1. HTML5

A quinta versão do HTML (HTML5) é uma especificação construída sobre o HTML/XHTML tradicionais, que visa atender de forma nativa as necessidades da Internet atual. Esta versão é fruto da junção de esforços de duas organizações: a *World Wide Web Consortium* (W3C) e a *Web Hypertext Application Technology Working Group* (WHATWG). Os objetivos principais a serem alcançados com o HTML5 são (Sanchez, 2011): (i) compatibilidade entre as novas funcionalidades e as anteriores versões do HTML, CSS e *JavaScript*; (ii) minimizar a necessidade de *plugins*; (iii) melhorar a gestão de erros; (iv) mais etiquetas nativas*;* (v) portabilidade, que permite igualar a forma como o conteúdo é visualizado independentemente do dispositivo.
O HTML5 acrescenta novos elementos sintáticos, nomeadamente, novas etiquetas de multimédia (*video*, *audio* e *canvas*), suporte para armazenamento local, etiquetas específicas de conteúdo (*article, footer, header, nav, section)* e novas etiquetas para formulários (*calendar, date, time, email, url, search).*
Com o lançamento do HTML5 passa a existir uma maior interatividade com os dados do lado do cliente. O mais frequente é recorrer ao uso de *plugins,* tal como *Adobe Flash, Microsoft Silverlight* ou *Scalable Vector Graphics* (Lubbers P. A., 2010). A utilização destas tecnologias limita os utilizadores a um grupo restrito de técnicas e empresas. Uma das soluções passa por calcular os dados previamente, do lado do servidor e, em seguida, enviá-los para o cliente (Corcoran, Mooney, Winstanley, &

Bertolotto, 2011). O que pode gerar um enorme trafego de rede e latência na comunicação.

Outra das novidades do HTML5 é a introdução da API *websocket*. A tecnologia *websocket — RFC 6455: The WebSocket Protocol*[8] — permite a criação de um canal de comunicação bidirecional, persistente e dedicado sobre um único *socket* TCP. Isto significa que a comunicação entre cliente e servidor pode ocorrer nos dois sentidos, em simultâneo e assincronamente, e que a ligação é mantida aberta até que uma das partes envie, explicitamente, um pedido de fecho de ligação (FETTE; MELNIKOV, 2011). O servidor pode comunicar com o cliente assim que os dados forem alterados, sem ser necessário esperar por qualquer pedido do cliente.

> *The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers.*
>
> *RFC 6455*

O protocolo *websocket* foi desenvolvido para ser utilizado especialmente em navegadores e servidores Web, e para dar suporte a aplicações Web que exigem comunicações em tempo real, tal como ilustra a Figura 3.1.

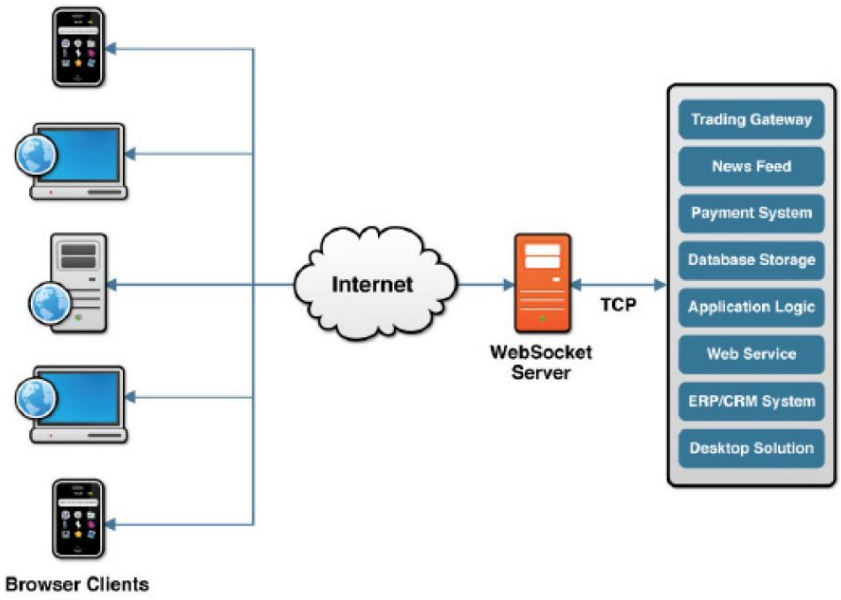


Figura 3.1: Servidor *websocket*

[8] Todas as regras que devem ser respeitadas para implementar um servidor ou cliente baseados no protocolo websocket estão descritas no *Request for Comments* (RFC), número 6455. Foi publicado em Dezembro de 2011, pela entidade IETF (*Internet Engineering Task Force*).

O protocolo *websocket* pode ser considerado como uma camada aplicacional sobre o TCP, adicionando: (i) um modelo de segurança; (ii) um esquema de nomes e endereçamento para suportar múltiplos serviços numa mesma porta e múltiplos nomes de servidores num mesmo endereço IP; (iii) um mecanismo de enquadramento; e (iv) um novo pedido de fecho. O objetivo é tornar o *websocket* o mais próximo possível do TCP padrão, mas tendo em conta os requisitos atuais da Web.

O *websocket* já é utilizado em vários sistemas e plataformas. São agora apresentados alguns exemplos que, para além de utilizarem o protocolo *websocket*, têm em comum o facto de permitirem a visualização de dados em tempo real ou de acederem remotamente a informação/ dispositivos:

Imagens 2D e 3D — sistema de visualização interativa 2D e 3D remota de dados médicos na Internet (Moraes, Amorim, Silva, & Pedrini, 2012); interação com gráficos 3D num navegador (Kapetanakis, Panagiotakis, & Malamos, 2013); modelo para aplicações de múltiplos ecrãs (Bassbouss, Tritschler, Steglich, Tanaka, & Miyazaki, 2013); transmissão e visualização de dados vectoriais geográficos (Corcoran, Mooney, Winstanley, & Bertolotto, 2011); sistema de troca de mensagens pictográfico para o SCALA - Sistema de Comunicação Alternativa para o Letramento de pessoas com Autismo (Ramos, 2013);

Acesso remoto — aplicações Web com monitorização, configuração e atualização remota (Furukawa, 2012); visualização de dados sem ser necessário fazer o seu *download* ou utilizar *hardware* específico (Wessels, Purvis, Jackson, & Rahman, 2011); acesso em tempo real a um sensor de vento (Pimentel & Nickerson, 2012).

Prospecção de dados — suporte a um sistema distribuído de *data mining* (Cassetti & Luz, 2011); plataforma *data binding* capaz de associar elementos do interface do utilizador com os seus respectivos dados (Heinrich & Gaedke, 2012);

Comunicação entre navegadores — implementação de um sistema de comunicações em tempo real entre navegadores, numa arquitetura *peer-to-peer* (WebRTC) e SIP (Quintana, 2013);

Sistemas baseados em eventos — sistema de comunicação seguindo o modelo publicação/ subscrição (Felício, 2012);

3.2. Protocolo websocket

O protocolo é definido na camada aplicacional e pode ser utilizado para superar restrições aplicadas à camada de transporte. É baseado no modelo de mensagens cliente-servidor e suporta a transmissão de dados binários ou texto simples.

3.2.1. Pedido inicial

Todas as ligações *websocket* começam com um pedido HTTP. Este pedido inicial (ou *handshake*) tem a particularidade de incluir no cabeçalho um pedido de atualização (ou *upgrade*). Isto indica que o cliente pretende atualizar a sua ligação para um protocolo diferente, neste caso, o protocolo *websocket*.
As várias fases de uma ligação *websocket* estão representadas na Figura 3.2, desde a criação de um canal TCP, passando pelo pedido de atualização do cliente, até ao encerramento da ligação.

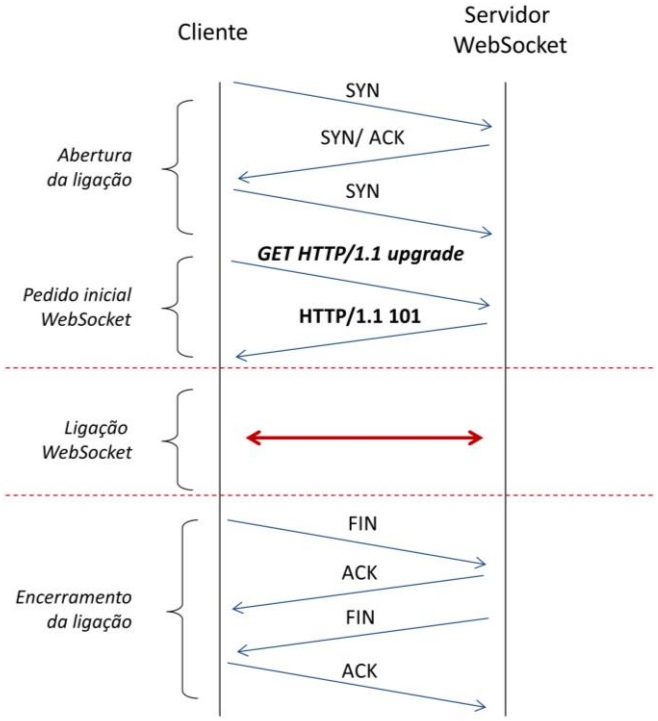


Figura 3.2: Etapas de uma ligação *websocket*

No cabeçalho existe também o campo *origin* que indica o URL do pedido *websocket*. Isto pode ser usado para distinguir ligações entre diferentes servidores e ligações feitas a partir de navegadores ou outro tipo de clientes. Não esquecer que este campo é apenas uma indicação, não muito segura, uma vez que clientes que não sejam navegadores podem colocar qualquer valor neste campo.

A Figura 3.3 representa o excerto de um pedido inicial por parte de um cliente. Até este pedido estar completo, a sessão *websocket* respeita o protocolo HTTP/1.1.

```
GET /echo HTTP/1.1
Host: echo.websocket.org
Origin: http://www.websocket.org
Sec-WebSocket-Key: 7+C600xYybOv2zmJ69RQsw==
Sec-WebSocket-Version: 13
Upgrade: websocket
```

Figura 3.3: pedido inicial HTTP de cliente para ligação *websocket*

A Figura 3.4 apresenta a resposta do servidor ao pedido inicial do cliente.

```
101 Switching Protocols
Connection: Upgrade
Date: Wed, 20 Jun 2012 03:39:49 GMT
Sec-WebSocket-Accept: fYoqiH14DgI+5ylEMwM2sOLzOi0=
Server: Kaazing Gateway
Upgrade: WebSocket
```

Figura 3.4: resposta HTTP do servidor

Para o pedido inicial ser completado com sucesso o servidor tem de provar ao cliente que recebeu o seu pedido de ligação e que compreende o protocolo *websocket*. O campo do cabeçalho *Sec-Websocket-Key* é enviado pelo servidor para o cliente para confirmar que está disposto para iniciar uma ligação *websocket*. Esta chave, com 20 *bytes* e codificados em *base-64*, resulta da concatenação de *Sec-Websocket-Key* (chave aleatória de 16 *bytes*, em *base-64*, enviada pelo cliente), com o Identificador Único Global fixo — "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" — definido no RFC 6455. Em seguida, é gerado o *hash SHA1* da sequência de caracteres anterior e codificado em *base-64*. A Listagem 3.1 exemplifica o processo de geração de uma chave de resposta, a partir da chave *Sec-WebSocket-Key*, enviada por um cliente.

Esta troca de chaves não oferece qualquer proteção aos clientes ou servidores que estabelecem uma ligação *websocket*, dado que são facilmente intersetáveis. Este mecanismo pretende proteger servidores que não suportam ligações *websocket* e eliminar a possibilidade de ataques de protocolos cruzados.

No pedido inicial são também incluídos os campos *Sec-WebSocket-Extensions, Sec-WebSocket-Protocol e Sec-WebSocket-Version*.

```
 Sec-WebSocket-Key:      "dGhlIHNhbXBsZSBub25jZQ=="
 Chave cliente + GUID:   "dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914
                         -47DA-95CA-C5AB0DC85B11"
 Hash SHA-1:             0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16
                         0x90 0xf6 0x46 0x06 0xcf 0x38 0x59 0x45
                         0xb2 0xbe 0xc4 0xea
Sec-WebSocket-Accept:    "s3pPLMBiTxaQ9kYGzzhZRbK+xOo="
```

Listagem 3.1: Geração da chave de resposta (*Sec-WebSocket-Key*)

3.2.2. Formato das mensagens

Enquanto uma ligação *websocket* se mantiver aberta, o cliente e o servidor podem trocar mensagens entre si em qualquer momento. O tráfego gerado é similar ao que é gerado através do HTTP ou HTTPS[9], utilizando o formato de parcelas (*frames*) binárias TCP. Geralmente só existe uma mensagem em cada parcela; porém, é possível que uma mensagem seja composta por várias parcelas. Os dados são transmitidos no formato de texto, codificado em UTF-8.

A Figura 3.5 representa o formato de uma mensagem *websocket*:

Opcode (tipo de mensagem) — 1 = Texto; 2 = Binária, 8 = Terminar ligação; 9 = Ping; 10 (hex 0xA) = Pong;

Mask — o primeiro *bit* do segundo *byte* indica se a mensagem está *mascarada*; o protocolo exige que os clientes mascarem todas as mensagens;

Length (tamanho da mensagem) — pode assumir três configurações:

< 126 *bytes* (ou 7 *bits*) = o tamanho está contido no segundo byte [0-125];

entre 126 e 216 *bytes* = são utilizados dois bytes extra [126-65536];

> 216 *bytes* = são utilizados 8 *bytes* do cabeçalho da mensagem.

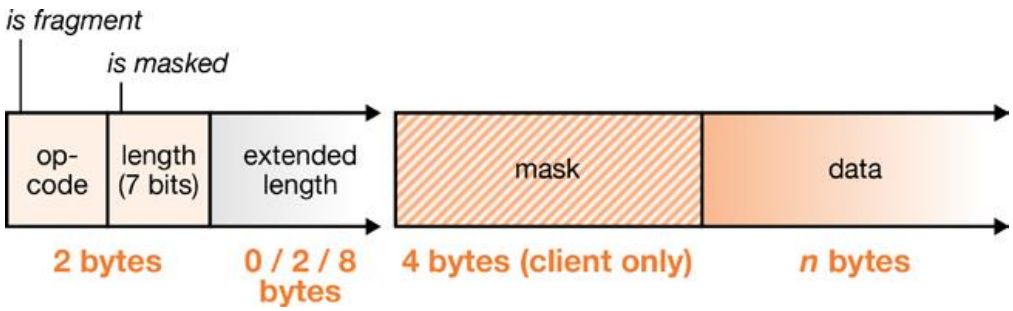


Figura 3.5: Mensagem *websocket*

[9] O protocolo HTTPS é uma extensão do protocolo HTTP que adiciona uma camada de segurança de nível inferior que utiliza o protocolo SSL/TLS. Esta camada permite o estabelecimento de um canal de comunicação seguro pela utilização de protocolos criptográficos e certificados digitais.

A técnica de mascarar as mensagens transforma o conteúdo de cada mensagem enviado pelo cliente fazendo XOR do byte *n* da mensagem com o módulo entre o *byte n e 4*, de um conjunto aleatório de 32 bits. O propósito é facilitar a passagem das mensagens por *proxies* transparentes (encobrindo, não cifrando, o conteúdo das mensagens) e evitar ataques de envenenamento da cache HTTP. Neste tipo de ataques o atacante assume controlo de uma memória cache HTTP alterando o seu conteúdo.

A partir da Figura 3.6 e Figura 3.7 é possível verificar o formato de duas mensagens *websocket*, do tipo texto (*opcode = 1*), a partir de um servidor e de um cliente. Na primeira mensagem é enviada informação no formato FIX, com a cotação de um valor mobiliário, neste caso, do Crédito Agrícola ("ACA"), da bolsa de Paris ("XPAR"). Na segunda mensagem é enviado um texto simples: "*Ola Muundo!*". Também é possível identificar as diferenças entre ambas. A primeira não mascara a mensagem, enquanto a segunda coloca o primeiro *bit* do segundo *byte* com o valor *1,* e a respetiva máscara nos quatro *bytes* seguintes. A primeira ocupou dois *bytes* adicionais para representar o tamanho dos dados da mensagem (entre 126 e 216 *bytes*), enquanto a segunda apenas precisou de um *byte*. O conteúdo da primeira mensagem apresenta-se em texto simples, enquanto na segunda é necessário descodificar o texto para aceder ao conteúdo correto.

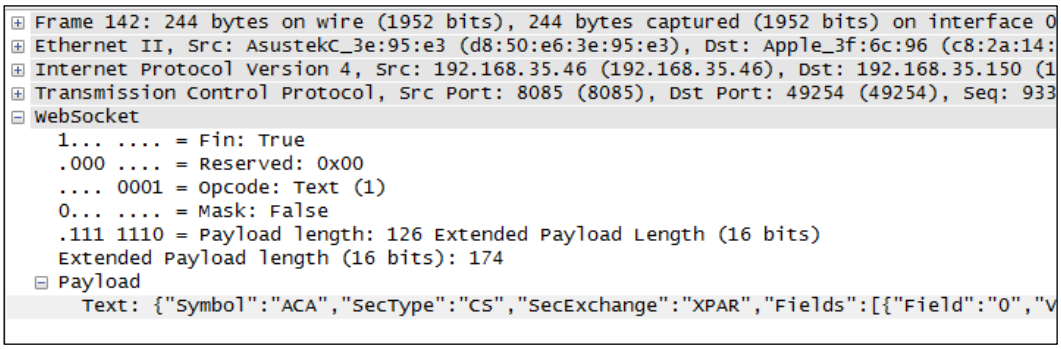


Figura 3.6: Detalhes de uma mensagem *websocket* enviada por um servidor

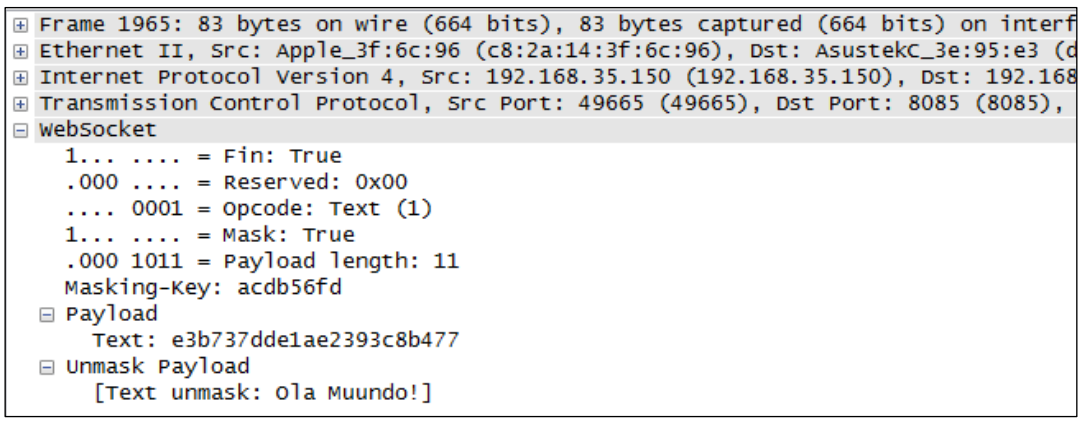


Figura 3.7: Detalhes de uma mensagem *websocket* enviada por um cliente

Foi utilizado o programa *Wireshark*[10] para realizar as capturas de ecrã.

3.3. API *websocket*

A API *websocket*, suportada pela maioria dos navegadores mais recentes, é uma interface que permite a aplicações Web o uso do protocolo *websocket*. Foi desenvolvida pela entidade *World Wide Web Consortium* (W3C). Esta API facilita a execução de operações como iniciar ou terminar uma ligação, enviar e receber mensagens e permanecer à escuta por eventos despoletados pelo servidor (Hickson, 2011).

3.3.1. Construtor

Para estabelecer uma ligação é necessário começar por criar uma nova instância de um objecto *websocket*. O protocolo *websocket* define dois esquemas URI[11] possíveis: ws:// (*websocket*) e wss:// (*websocket* sobre *SSL/TLS*), para tráfego entre cliente e servidor não encriptado e encriptado, respectivamente. Para a ligação normal é utilizado o HTTP, para a ligação segura é utilizado o HTTPS.

```
var ws = new WebSocket("ws://www.websocket.org");
```

Listagem 3.2: Construtor *websocket*

O construtor recebe o URL do servidor enquanto argumento obrigatório — Listagem 3.2 —, e pode ainda incluir um argumento opcional: um ou mais protocolos que o servidor deverá incluir na resposta de forma a estabelecer ligação (por exemplo: XMPP, SOAP, STOMP).

A Listagem 3.3 ilustra um exemplo de um construtor *websocket* que inclui como argumentos o endereço do servidor e uma lista de protocolos (que o cliente suporta). Cabe ao servidor escolher o subprotocolo[12] que irá usar para estabelecer a ligação.

```
var echoSocket = new WebSocket("ws://echo.websocket.org",
     ["com.kaazing.echo","example.imaginary.protocol"])

echoSocket.onopen = function(e) {
     console.log(echoSocket.protocol);
}
```

Listagem 3.3: Construtor *websocket* com suporte a protocolos

[10] www.wireshark.org - *Wireshark is the world's foremost network protocol analyzer. It lets you see what's happening on your network at a microscopic level. It is the de facto (and often de jure) standard across many industries and educational institutions.*

[11] Um Identificador Uniforme de Recursos (URI) é uma sequência de caracteres compacta usada para identificar ou denominar um recurso na Internet.

[12] *RFC 6455: The Websocket Protocol*, secção 1.9. "*Subprotocols Using the WebSocket Protocol*"

3.3.2. Eventos

A API *websocket* é baseada no modelo de eventos. O que significa que a aplicação tem apenas de se manter à escuta pela ocorrência de eventos ou mensagens nos objetos *websocket* e mudanças de estado na ligação.

Os objetos *websocket* geram quatro tipos de eventos:

- `Open` — informa que a ligação foi estabelecida com o servidor e que a comunicação de mensagens pode começar;
- `Message` — sempre que uma mensagem é recebida;
- `Error` — quando ocorre alguma falha; geralmente a ligação termina depois deste evento;
- `Close` — quando a ligação é terminada.

A Listagem 3.4 ilustra um exemplo de utilização das funções de retorno, correspondentes a cada um dos eventos atrás referidos.

```
function setup() {
    ws = new WebSocket("ws://echo.websocket.org/echo");

    ws.onopen = function(e) {
          log("Ligação estabelecida!");
    }

    ws.onclose = function(e) {
          log("Ligação terminada: " + e.reason);
    }

    ws.onerror = function(e) {
          log("Erro!");
    }

    ws.onmessage = function(e) {
          log("Mensagem recebida: " + e.data);
    }
}
```

Listagem 3.4: Eventos *websocket*

3.3.3. Métodos

Depois de estabelecida a ligação entre cliente e servidor é possível invocar os métodos: (i) ***send(data)*** — para enviar mensagens de texto ou dados binários para o servidor enquanto a ligação está ativa; e (ii) ***close()*** — para terminar uma ligação. A Listagem 3.5 ilustra um exemplo de utilização destes dois métodos.

```
// Enviar uma mensagem de texto
ws.send("Hello World!");

// Terminar uma ligação
ws.close();
```

Listagem 3.5: Métodos *websocket*

3.3.4. Atributos dos objetos websocket

A API *websocket* disponibiliza três atributos dos objetos *websocket* que podem fornecer informação útil acerca do seu estado:

`readyState` — informa o estado da ligação (*connecting, open, closing, closed*);

`bufferedAmount` — se for necessário transferir grandes quantidades de informação, este atributo indica a quantidade de bytes que ainda não foram enviados para o servidor;

`protocolo` — este atributo contém o nome do protocolo escolhido pelo servidor durante o pedido inicial.

3.4. Segurança

O protocolo *websocket* resolve problemas de transporte, não de segurança. Pelo que, nesta secção, vamos enumerar um conjunto de recomendações práticas para a implementação de serviços e aplicações que utilizem *websocket*, de uma forma segura.

Em aplicações Web a segurança é geralmente garantida através da encriptação da camada de transporte e da política de segurança de origem comum, implementada em navegadores Web (Erkkilä, 2013). A encriptação através do protocolo *Transport Layer Security* (TLS) assegura um canal de comunicação seguro, enquanto a política da origem comum protege o utilizador contra referências a páginas Web maliciosas.

No contexto dos navegadores Web, a política de origem comum restringe a interação entre documentos ou a execução de pequenos pedaços de código se tiverem origens diferentes. Duas páginas têm a mesma origem se o protocolo, a porta e o servidor forem iguais. O mecanismo ganha particular importância em aplicações Web atuais que fazem uso intensivo de *cookies* para guardar informações dos utilizadores, sejam elas de simples autenticação ou informações pessoais confidenciais.

As aplicações Web que utilizem *websocket* possuem as mesmas vulnerabilidades relativamente a outras aplicações. Ou seja, se é possível examinar ou intersetar mensagens HTTP, também é possível examinar ou interceptar mensagens *websocket*. O que significa que podem ser utilizados os mesmos métodos para ler mensagens ou injetar informação maliciosa. Por exemplo, ataques como *Man in the Middle* (MITM) ou *Cross Site Scripting* (XSS), continuam a constituir uma ameaça.

O ataque MITM (atacante no meio) acontece quando os dados trocados entre duas partes são interceptados, registados e possivelmente alterados pelo atacante, sem que ambas as partes se apercebam.

Um ataque XSS explora vulnerabilidades no código HTML de páginas ou aplicações Web. O ataque consiste em injetar endereços ou pequenos pedaços de código (normalmente em *JavaScript* ou *VBscript*) do lado da vítima, de forma a recolher dados privados desta. O código inserido pode, entre outras coisas, redirecionar o utilizador para uma página falsificada, disponibilizar ao atacante cookies ou identificadores de sessão, ou fazer com que o navegador execute alguma ação danosa.

A primeira forma de proteger uma aplicação Web contra ataques XSS é garantir que a aplicação valida todos os dados que recebe como entrada, incluindo cabeçalhos, campos de formulários, *cookies*, palavras de consulta e campos escondidos.

A autenticação de clientes por parte do servidor também recorre a mecanismos iguais ao HTTP, tal como uso de *cookies*, autenticação HTTP ou TLS.

Porém, existem diferenças entre o protocolo *websocket* e o HTTP, o que tem implicações na segurança da Web. Por exemplo, as políticas de origem são diferentes e as mensagens *websocket* não incluem cabeçalhos HTTP, o que pode afectar o comportamento de *firewalls* e *proxies*.

3.4.1. Analisar o cabeçalho: origin

O protocolo *websocket* utiliza o modelo de segurança de origem, definido em RFC 6454, para estabelecer novas ligações. Pelo que o servidor deverá estar configurado para o controlo de acesso baseado na origem, de forma a garantir ligações seguras de origens diferentes.

A API *websocket*, do lado do cliente, permite que uma aplicação Web inicie uma nova ligação e envie dados para qualquer servidor. Esta característica facilita a partilha de recursos entre diferentes serviços.

Do lado do servidor é recomendável que seja sempre verificada a origem da mensagem, através do campo *origin* do cabeçalho, e decidir se essa ligação é aceite ou rejeitada. Este processo é importante para impedir ataques *Denial of Service* (DoS). Este tipo de ataques tem como objectivo impedir que uma página Web, um

determinado serviço ou um servidor completo deixe de estar disponível temporária ou indefinidamente.

De salientar que esta política de verificação da origem da mensagem não impede ninguém de se conectar a um servidor, uma vez que o cabeçalho pode ser facilmente alterado. Porém, este mecanismo protege o cliente de ataques *Cross-site request forgery* (CSFR).

Este tipo de ataques, que deriva do ataque XSS, induz o utilizador a executar ações indesejadas numa aplicação Web em que está autenticado. Com o auxílio de alguma *engenharia social*, o atacante herda a identidade e os privilégios do utilizador, e executa operações sem este se aperceber, tal como mudar o endereço de email, a morada ou *password*, ou mesmo executar uma transação bancária.

3.4.2. Utilizar wss

A comunicação através do protocolo ws é feita em texto simples. Portanto, é sempre preferível o uso do protocolo seguro wss (*websocket* sobre SSL/TLS). Tal como o protocolo HTTPS, o wss é encriptado, protegendo desta forma a ligação de vários tipos de ataques, nomeadamente, MITM.

O protocolo SSL/TLS tem como principal objetivo proporcionar privacidade e integridade dos dados na comunicação entre dois terminais. É composto por duas camadas, a *TLS Record Protocol* e a *TLS Handshake Protocol*.

O primeiro protocolo proporciona segurança ao nível do canal de comunicação. A comunicação é privada e confiável. A privacidade é obtida através da encriptação dos dados pela utilização de criptografia simétrica (AES, RC4, entre outros). Esta é baseada na partilha de um segredo (uma chave simétrica) que somente ambas as entidades envolventes devem ter conhecimento. Contudo, a criptografia simétrica tem um problema associado, no ato da partilha do segredo (antes da transferência de qualquer *bit* de dados) a comunicação pode ser interceptada por terceiros que podem usufruir do segredo que agora é partilhado por três entidades. Portanto, de modo a ultrapassar este problema, por cada comunicação que é estabelecida são geradas chaves para a encriptação simétrica, baseadas na negociação de um segredo assegurada pelo protocolo *TLS Handshake* (Wang, Salim, & Moskovits, 2013).

A integridade dos dados é mantida pela utilização de *Message Authentication Codes* que usufruem de funções *hash* como o SHA-1.

É então função do protocolo *TLS Handshake* permitir a autenticação de ambas as entidades para que possam negociar um algoritmo de encriptação, assim como as chaves criptográficas, antes de ser transmitida qualquer tipo de informação entre os envolventes (Dierks & Rescorla, 2008).

Com o protocolo HTTPS, o cliente e o servidor começam por estabelecer uma ligação segura e só depois começam o protocolo HTTP. Da mesma forma, o wss estabelece primeiro uma ligação segura, depois inicia o pedido inicial HTTP e, finalmente, a atualização da ligação para o protocolo *websocket*.

Na Figura 3.8 é possível verificar que tal como o HTTPS é o equivalente a transportar HTTP sobre TLS, wss é o equivalente a transportar WS sobre TLS.

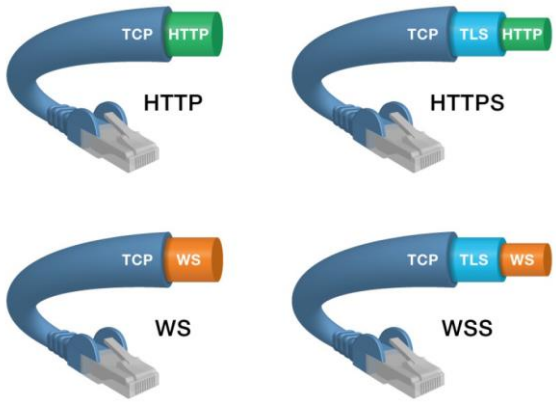


Figura 3.8: HTTP, HTTPS, WS e WSS

A utilização do protocolo TLS garante confidencialidade, integridade e disponibilidade à comunicação *websocket*. Também melhora a capacidade de transmitir informação através de intermediários e *proxies*.

Atualmente, a API *websocket* proíbe o uso de *ws* a partir de um recurso que foi obtido com HTTPS.

3.4.3. Validação dos dados do cliente e servidor

As ligações *websocket* podem ser estabelecidas sem um navegador Web. Pelo que, do lado do servidor, deve sempre existir uma validação dos dados provenientes do cliente, antes de serem executados. Por exemplo, ataques como a injeção de SQL são igualmente possíveis em *websocket* como são através de HTTP.

Do lado do cliente, essa verificação também deverá ocorrer. Em particular, verificar o formato de datas, conjuntos de caracteres e codificações permitidas, linhas de quebra, formulários de acesso ao sistema, validação extensa de todos os dados enviados pelo cliente. Por exemplo, o cliente não deve executar a informação recebida pelo servidor diretamente no DOM, sem uma validação prévia; ou ainda, se a resposta do servidor seguir o formato JSON deverá ser utilizada a função *JSON.parse()* para desserializar os dados corretamente.

3.4.4. Autenticação e autorização

O protocolo *websocket* não disponibiliza qualquer mecanismo de autorização ou autenticação. Pelo que devem ser utilizados os mesmos processos de autenticação que são utilizados noutras aplicações Web, nomeadamente, autenticação HTTP ou através de *cookies*.

Este processo de autorização é dificultado pelo facto de, frequentemente, existir um servidor que responde aos pedidos HTTP e outro que trata dos pedidos *websocket*. Para resolver este problema é apresentado em (Heroku Dev Center, 2014) uma solução de um sistema de autenticação baseado na troca de bilhetes: (i) o cliente começa por requisitar um bilhete de autorização ao servidor HTTP; (ii) o servidor cria um bilhete com base em alguma identificação, no IP do cliente, um *timestamp* e mais algumas informações; (iii) o servidor guarda o bilhete criado e envia-o para o cliente; (iv) o cliente abre a ligação *websocket* e envia juntamente o bilhete recebido no pedido inicial; finalmente, (v) o servidor antes de aceitar a ligação *websocket*, compara a informação deste bilhete com o que tem guardado, nomeadamente, o endereço IP do cliente, se o bilhete já foi utilizado ou se já expirou, e mais algumas verificações de permissões.

3.4.5. Validação do subprotocolo

O protocolo *websocket* foi pensado como uma camada de transporte para outros protocolos (tal como o TCP), mas para a Web. Por exemplo, é possível transportar protocolos como XMPP, AMQP ou STOMP, pela Web, através de *firewals* e *proxies*, e utilizando as portas normais 80 ou 443.

No cabeçalho de uma mensagem *websocket* existe o campo *Sec-WebSocket-Protocol* que indica o nome do subprotocolo, ou seja, o protocolo da camada da aplicação que será utilizado sobre o protocolo *websocket*. Neste campo podem ser indicados vários protocolos e cabe ao servidor escolher o mais indicado.

Depois de definido o subprotocolo, o servidor deverá adequar as suas políticas de segurança a este subprotocolo, intersetar o tráfego e verificar a validade do seu conteúdo. Esta inspeção ao nível do subprotocolo permite aumentar o nível de segurança, pois, vai para além da análise de pacotes HTTP.

3.4.6. Limitar o número de ligações

Se o servidor não estiver a aceitar ligações suficientemente rápido, o cliente não deverá tentar criar novas ligações, até que a anterior seja aceite. Por outro lado, o servidor deverá monitorizar constantemente vários parâmetros, como sejam, o consumo de recursos computacionais e o número de falhas que vão ocorrendo.

Desta forma é possível evitar: ataques como *DoS*, a perda de ligações ativas, a degradação da disponibilidade e segurança do servidor.

3.4.7. Evitar comunicação via túnel

É possível criar uma ligação TCP via túnel através de *websocket*. Por exemplo, pode-se criar uma ligação direta entre uma base de dados e um navegador Web. Porém, este procedimento deve ser evitado, dado que expõe este serviço a vários tipos de perigos, desde ataques XSS até a um controlo remoto total do serviço por parte do atacante.

A partir da Tabela 3.1 é possível resumir e identificar as principais medidas de segurança que devem ser adoptadas por aplicações ou serviços Web que utilizem *websocket*, perante os principais ataques informáticos atuais.

Tabela 3.1: Ataques ao protocolo e API *websocket* e respectivas medidas de segurança

Tipo de ataque	**Medidas de segurança**
Man-in-the middle	Utilizar *wss* (*websocket* secure)
Denial of service, Cross Site Request Forgery	Conferir o cabeçalho: *origin*, Não criar nem aceitar novas ligações se as anteriores não tiverem sido corretamente iniciadas ou concluídas
Proxy server attacks, cache poisoning	Camuflar mensagens (*masking*)
Cross Site Scripting, SQL Injection	Conferir o cabeçalho: *Sec-WebSocket-Accept* , Validar sempre os dados do cliente e servidor, Evitar comunicação via túnel
Subprotocolos	Adequar as políticas de segurança do servidor de acordo com subprotocolo escolhido, intersetar o tráfego e verificar a validade do seu conteúdo

Neste capítulo foi apresentado o protocolo *websocket*, em particular, o processo de criação e as características de uma ligação *websocket*, o pedido inicial e o formato das mensagens *websocket*. Em seguida, o construtor, os eventos e os métodos da API *websocket*, que permite a utilização do protocolo, foram descritos. Finalmente foi enumerado um conjunto de recomendações práticas para o desenvolvimento de serviços e aplicações Web seguras, baseadas no protocolo *websocket*.

4. Apresentação do protótipo

Este capítulo inclui a descrição da arquitetura e das principais funcionalidades do protótipo desenvolvido ao longo do estágio. Na primeira secção é apresentada a tecnologia escolhida para elaborar este protótipo, tendo em conta as várias alternativas apresentadas e os objectivos que se pretendem atingir. Em seguida é abordada a arquitetura do *software*, a forma como o código está organizado e a descrição detalhada das classes implementadas mais relevantes. Na terceira secção é apresentado o protocolo FIX e o formato JSON. Finalmente são enumeradas as etapas necessárias para testar o protótipo e analisadas as mensagens trocadas através de algumas capturas de ecrã.

4.1. Escolha da tecnologia

Tendo em conta as vantagens e inconvenientes das várias alternativas apresentadas nos capítulos 2 e 3, foi escolhido o protocolo *websocket* para desenvolver o novo servidor — SDW. Nesta escolha foram decisivos os seguintes critérios: utilização eficiente da largura de banda, escalabilidade, comunicação bidirecional, disponibilidade em múltiplas plataformas e linguagens sem a necessidade de recorrer à instalação de código externo, latência, independência de qualquer serviço externo (nomeadamente IIS), performance e simplicidade na arquitetura do sistema.
Na Tabela 4.1 estão listadas as plataformas dos atuais clientes do SDW. Podemos confirmar que existe pelo menos uma implementação do protocolo *websocket* disponível para atualizar todos os clientes no futuro (que não são navegadores Web).

Tabela 4.1: Lista dos clientes do SDW com suporte *websocket* nativo e não nativo

Nativo (em pelo menos uma versão do protocolo)	Java 7 (JSR 356)	docs.oracle.com/javaee/7/tutorial/doc/websocket.htm oracle.com/technetwork/articles/java/jsr356-1937161.html
	.NET 4.5	msdn.microsoft.com/en-us/library/system.net.websockets.websocket.aspx
	Navegadores Web	Internet Explorer (a partir da versão): 10 Firefox (PC & Android): 11 Chrome (PC & Mobile): 16 Safari (Mac, iOS): 6 Opera (PC & Mobile): 12.10
Não nativo	Objective-C	github.com/square/SocketRocket
	Java 1.5/ Android 1.6 (API 4)	github.com/TooTallNate/Java-WebSocket http://java-websocket.org

A partir da análise de vários estudos comparativos entre o protocolo *websocket* e outras tecnologias, destacamos agora algumas das principais vantagens do protocolo *websocket* relativamente às alternativas:

Performance e largura de banda — vários estudos demonstram como o protocolo *websocket* pode aumentar a performance e reduzir a utilização de largura de banda, em relação a técnicas de *polling* e *HTTP streaming* (Pierro, Cavallari, Guida, & Innocente, 2010) (Heinrich & Gaedke, 2012) (Laine & Säilä, 2012) (Varela, 2011) (Lubbers & Greco, 2012) (Corcoran, Mooney, Winstanley, & Bertolotto, 2011) (Sheiko, 2012);

Adequado para sistemas distribuídos e heterogéneos (Lopes, 2012);

Comunicação bidirecional numa única ligação — em técnicas alternativas é necessário a manutenção de duas ligações em simultâneo, o que gera um elevado consumo de recursos computacionais e complexidade na arquitetura do sistema (Lubbers & Greco, 2012);

Escalabilidade — testes de performance de uma aplicação Web que recebe dados em tempo real, demonstram a escalabilidade do protocolo *websocket* (Qveflander, 2010);

Suporte nativo — o protocolo *websocket* é atualmente a técnica melhor suportada, de forma nativa, pelos principais navegadores Web (Sheiko, 2012);

Consumo eficiente de recursos computacionais — comparativamente à técnica de *long-polling* e SSE, o protocolo *websocket* faz a melhor gestão de memória e utilização de CPU (Sheiko, 2012);

Vantagens a todos os níveis, em relação à utilização do protocolo BOSH ou da técnica de *XHR-polling* (Govaerts, et al., 2011) (Agarwal, 2012).

4.2. Implementação e arquitetura do software

De forma a diminuir a latência na comunicação e aumentar a coerência de dados entre o *ServerDeal* e as aplicações cliente *SifoxDealWeb* e *SifoxDealMobile* foi desenvolvido um protótipo do SDW de raiz, em C++11[13], baseado no protocolo de comunicação *websocket*.

A Figura 4.1 apresenta a estrutura geral do protótipo que foi desenvolvido. O novo SDW recebe mensagens do *ServerDeal*, de acordo com o protocolo FIX, utilizando *stream sockets* TCP. O componente *ServerDeal* foi adaptado e simplificado de forma a enviar ininterruptamente um conjunto fixo de 65 títulos[14] das bolsas de valores de

[13] C++11 é o nome da linguagem C++ padrão, aprovada em 2011.
[14] Lista do código dos títulos que foram subscritos pelo SDW: BCP, EDPR, ALTR, BES, BPI, BRI, GALP, JMT, EGL, PTC, CFN, SEM, PTI, SNC, RENE, AI, GLE, AIR, ALO, CS, BNP, ACA, RNO, OR, ML, VIV, FP, CA, VK, EN, LG, SAN,

Lisboa, Paris, Nova Iorque e NASDAQ. Em seguida, envia esta informação para todos os clientes que estão ligados ao servidor. As mensagens cumprem o formato JSON e o protocolo de comunicação utilizado é o *websocket*. O cliente utiliza um navegador Web para visualizar os dados que chegam do SDW. Está implementado em HTML/ *JavaScript* e utiliza a API *websocket* para estabelecer a ligação.

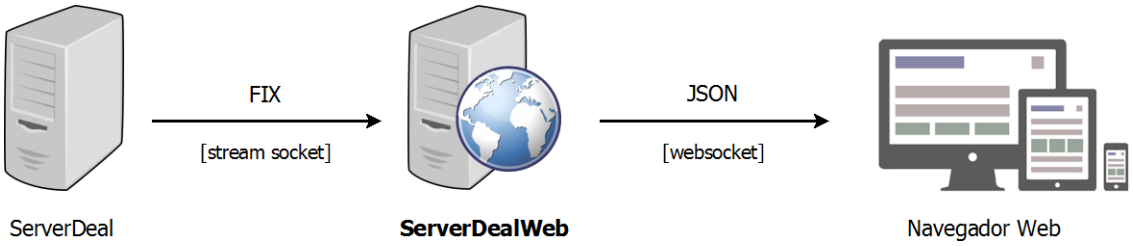


Figura 4.1: Esquema geral do protótipo

Partindo do modelo cliente-servidor, a adopção desta arquitetura tem como objectivos garantir às aplicações cliente *SifoxDealWeb* e *SifoxDealMobile*:

Escalabilidade — dado que o número de clientes ligados em simultâneo varia constantemente, podendo existir longos períodos de tempo com um número elevado de ligações;

Integridade — um dos requisitos mais importantes, uma vez que se trata de uma aplicação de negociação em bolsa que envolve operações críticas como transações de volumes de capital e troca de informações confidenciais;

Mobilidade — enquanto processo em execução, o servidor pode ficar alojado na mesma máquina do processo cliente, num ponto da rede local ou alojado numa máquina remota, a uma longa distância do processo cliente.

A partir da Figura 4.2 é possível identificar as principais classes do SDW e respetivas relações.

4.2.1. Classe `wsServer`

A classe principal é `wsServer` *(websocket server)*. Esta é a classe responsável por executar os métodos iniciais para arrancar o servidor, estabelecer a ligação com o *ServerDeal*, criar e gerir todas as novas ligações que são estabelecidas.

BN, RI, SU, SAF, PUB, VIE, CAP, MC, TEC, FB, TSLA, AMZN, INTC, AAPL, MSFT, GOOG, CSCO, COV, MDT, FIO, BAC, RAD, WMD, WFT, GE, AMD, PFE, SUNE, PBR, FSTF, NCT, EXC, TWTR.

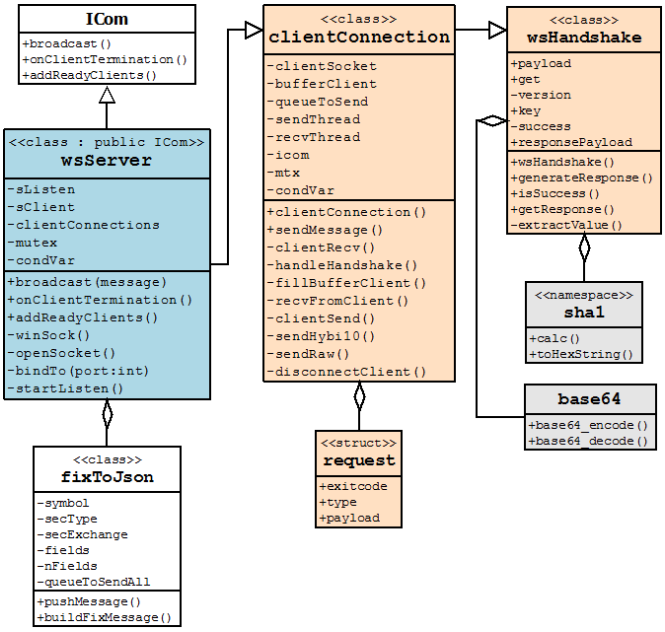


Figura 4.2: Estrutura geral do SDW

A classe utiliza uma *map* – *std::map<Key, Data, Compare, Alloc>* para guardar as instâncias dos objetos da classe *clientConnection* que são criadas sempre que ocorre uma nova ligação. Esta estrutura de dados associa um objeto do tipo *Key* (a chave) a um objeto do tipo *Data* (o valor). No protótipo as chaves são objetos da classe *clientConnection* e o valor associado ainda não está definido. Mas no futuro será útil para distinguir diferentes tipos de ligações/ clientes.

O método *broadcast()* envia uma mensagem para todos os clientes que estão ligados ao servidor. No caso de alguma ligação terminar, o método *onClientTermination()* remove o cliente da lista de clientes ligados e apaga a respetiva instância. O método addReadyClients() adiciona clientes a uma lista de clientes que estão prontos para comunicar.

Todas as operações sobre a lista de clientes são sincronizadas através de variáveis de exclusão mútua (*mutex*). Estas variáveis atuam como um semáforo onde, em cada instante, apenas um *thread* ou processo consegue aceder a um recurso partilhado.

Um *thread* é uma execução sequencial de um programa. Cada *thread* tem a sua própria pilha, prioridade e conjunto de registos virtuais. Os *threads* subdividem o comportamento de um programa em subtarefas independentes.

Para arrancar o servidor é criada uma instância de um objeto da classe *wsServer*. A Figura 4.3 ilustra o fluxo de informação desta classe.

O primeiro conjunto de operações utiliza várias funções da API *Winsock* do *Windows*. Esta API permite o acesso a protocolos de transporte, a aplicações baseadas em *Windows*.

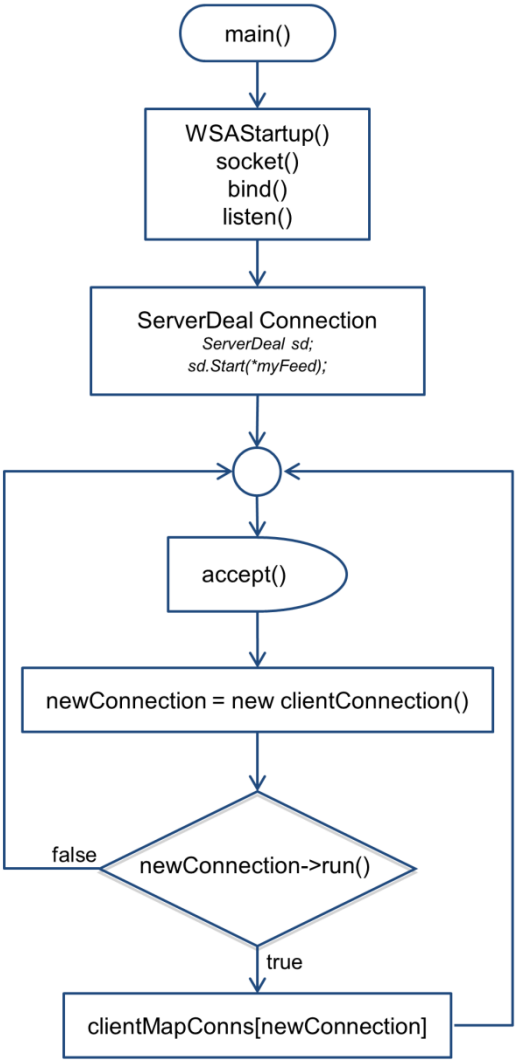


Figura 4.3: Gráfico de fluxo da classe *wsServer*

A primeira função a ser executada é `WSAStartup()`[15]. Em seguida é criado um novo *socket*, através da função `socket()`, que serve para ficar à escuta de novas ligações — `sListen`. Se não ocorrer nenhum erro é devolvido um inteiro positivo, ou seja, um descritor para referenciar o novo *socket* — `sClient`. Este descritor é necessário em todas as operações subsequentes.

Antes de se poder receber ou enviar informação através de um *socket* é necessário associá-lo a um endereço IP e a uma porta[16]. Esta operação é executada através da função `bind()`.

A próxima função a ser chamada — `listen()` — coloca o *socket* criado num estado de escuta. A partir de agora o protótipo poderá receber a ligação de outros *sockets*.

[15] Esta função inicializa o uso da biblioteca *WS2_32* por um processo e verifica se o computador suporta a versão do *Winsock* a ser utilizada.
[16] Uma porta é um número que associa os pacotes recebidos da rede, com um determinado processo em execução nessa máquina.

A seguir é feita a ligação ao *ServerDeal*, através da criação de uma instância de um objecto da classe `ServerDeal`. Esta classe foi originalmente desenvolvida pela Finantech. A versão que foi incluída no protótipo é uma adaptação do que existe atualmente na *plataforma Sifox*.

Depois deste conjunto de funções para iniciar o servidor, a execução do programa fica pendente na função `accept()`, que recebe como argumento o *socket* criado. Esta função permite aceitar um pedido de conexão, devolver um novo *socket* já ligado ao emissor do pedido e o *socket* original continuar em escuta. A função `accept()` permite ao servidor aceitar e gerir múltiplas ligações em simultâneo.

Sempre que ocorre uma nova ligação é criada uma instância de um objeto da classe `clientConnection`, cuja referência é guardada na `map clientConnections`.

4.2.2. Classe clientConnection

Para cada ligação é criada uma instância de um objeto da classe `clientConnection`. Esta classe permite a troca de mensagens entre o servidor e o cliente, baseadas no protocolo *websocket*. A cada cliente está associado um *socket*, uma fila de envio (`queueToSend`[17]), um vector que armazena temporariamente os dados que são recebidos do cliente (`bufferClient`) e dois *threads (*`sendThread` *e* `recvThread`*)*, que são criados ao iniciar a nova instância do objecto da classe `clientConnection`, tal como ilustra a Figura 4.4.

O primeiro *thread* — `recvThread` — recebe o pedido inicial do cliente, constrói a respectiva resposta segundo o protocolo *websocket* e envia-a para o cliente. Se esta troca inicial for bem sucedida é criada uma ligação *websocket*. Este *thread* permanece bloqueado, à espera de dados enviados pelo cliente, na função `recv()`. Depois de uma mensagem ser recebida, é descodificada segundo o protocolo *websocket* e feita a verificação do seu tipo. Se for do tipo *texto*, a mensagem é colocada na fila de envio de todos os outros clientes ligados ao servidor (através do método `icom.broadcast`) e é enviada uma notificação para o segundo *thread*. Se for uma mensagem em formato binário ou *ping* é feito um eco da mensagem recebida para o mesmo cliente ou enviada uma mensagem do tipo *pong, respetivamente*.

Para todos os outros tipos, o cliente é desconectado: o outro *thread* é terminado corretamente e o cliente é removido da lista de ligações ativas do servidor.

O segundo *thread* — `sendThread` — aguarda pela entrada de dados na fila de envio. Os dados quando chegam à fila `queueTosend` são copiados para uma fila temporária

[17] Em C++ a forma de representar um *byte* é utilizar o tipo `unsigned char`. Por esta razão, a variável `queueToSend` é declarada da seguinte forma: `queue<vector<unsigned char>>`; ou seja, uma fila de conjuntos de *bytes*/ mensagens websocket.

e retiradas da anterior. Em seguida, a segunda fila é percorrida e todos os conteúdos são enviados para o cliente seguindo o formato *websocket*.

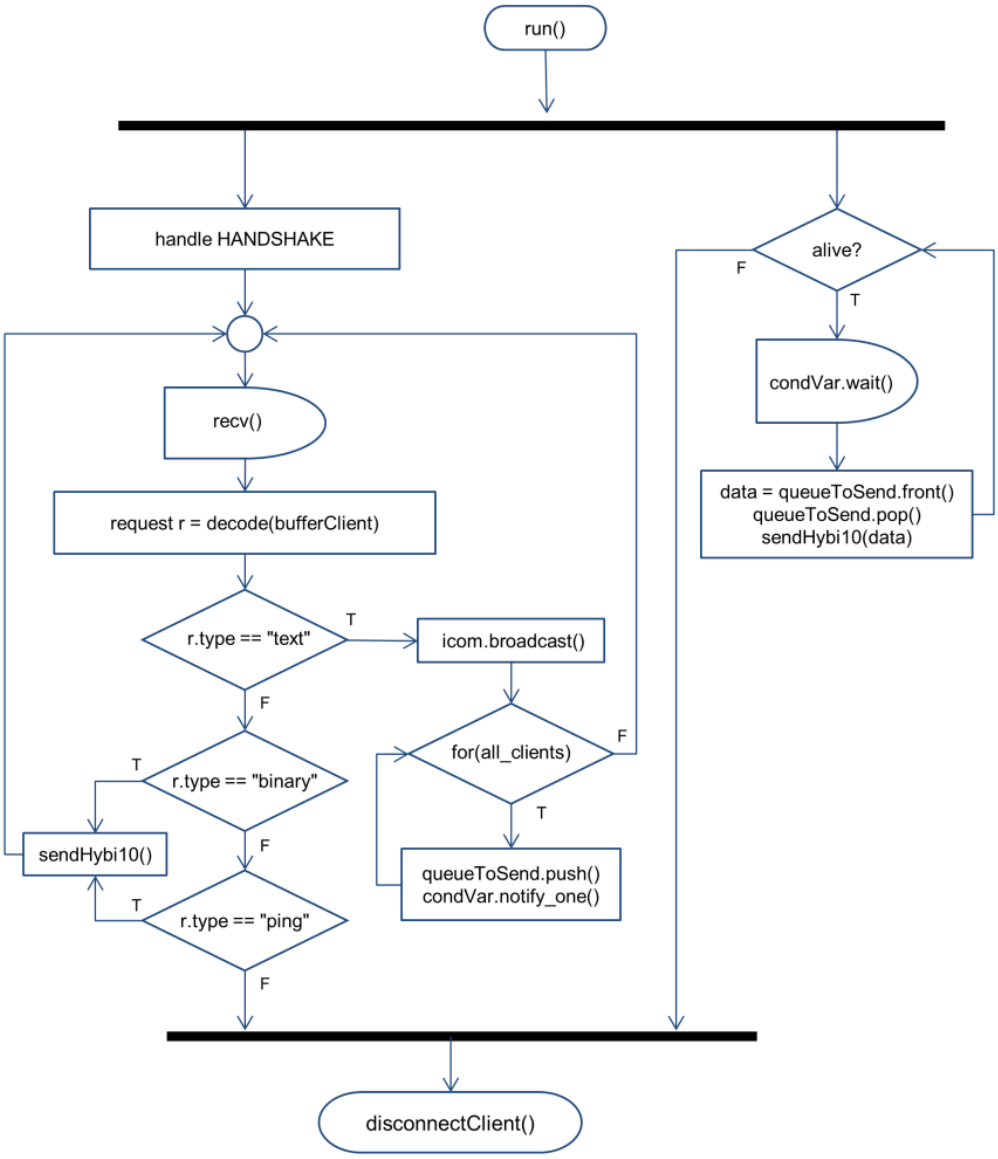


Figura 4.4: Gráfico de fluxo da classe *clientConnection*

A utilização de duas filas permite libertar rapidamente a fila `queueTosend` para voltar a ser usada pelo primeiro *thread*. Desta forma pretende-se evitar que a utilização da fila `queueTosend` fique interdita durante todo o processo de envio, uma vez que este processo pode demorar um certo tempo ou ocorrer algum erro a enviar.

A divisão em dois *threads* dos processos de receção e envio dos dados do cliente torna a gestão da informação mais eficiente, dado que a disponibilidade de envio ou receção de dados é garantida independentemente da frequência e do tamanho das mensagens. Por exemplo, numa situação em que é necessário enviar uma mensagem

de grandes dimensões do servidor para o cliente, se existisse apenas um *thread*, durante o tempo de envio da mensagem o cliente não poderia receber novos dados.

A comunicação entre os dois *threads* e a sincronização do uso da fila de envio é feita através de variáveis de condição e de exclusão mútua em conjunto. Desta forma fica assegurado o acesso exclusivo à fila de envio.

As variáveis de condição são utilizadas para suspender a execução de um *thread* até que: (i) uma notificação seja recebida; (ii) um tempo limite expirar; ou (iii) uma falsa notificação ocorra. Qualquer segmento que pretende esperar em `std::condition_variable` tem de adquirir primeiro o `std::unique_lock`. As operações de espera libertam o *mutex* atomicamente e suspendem a execução do *thread*. Por outro lado, quando a variável de condição é notificada o *thread* desperta e o *mutex* é readquirido.

Quando algum dos dois *threads* termina, a referência à instância da classe é removida da lista de ligações ativas e é chamado o destrutor da classe `clientConnection`. O destrutor fecha o *socket* da ligação e executa as funções `join()` e `detach()`, em ambos os *threads* alternadamente — `recvThread` e `sendThread` —, consoante se for o primeiro ou o segundo a encerrar.

Enquanto a função `join()` espera que o *thread* associado finalize, a função `detach()` separa-se do *thread* associado, tornando-se independente do processo que o criou.

4.2.3. Classe `ICom`

Em programação orientada a objetos um membro de uma classe do tipo `virtual` pode ser redefinido em classes derivadas.

A classe `ICom` declara um conjunto de métodos (`broadcast`, `onClientTermination`, `addReadyClients`) do tipo `virtual`, que são definidos na classe `wsServer`.

Esta organização permite: (i) a gestão de todas as ligações numa única instância do objecto da classe `wsServer`; (ii) que todas as instâncias `clientConnection` consigam comunicar entre si; e (iii) que os clientes sejam corretamente encerrados e removidos da lista de clientes ativos, quer ocorra ou não algum erro.

4.2.4. Outras classes

Agregada à classe `wsServer` existe a classe `fixToJson` que serve para construir as mensagens no formato JSON que são enviadas para os clientes. Esta classe utiliza a

biblioteca *rapidjson*[18] que fornece um conjunto de métodos para construir mensagens no formato JSON.

A classe `wsHandshake` trata do pedido inicial de cada cliente, através dos métodos `getResponse()` e `generateResponse()`. Esta classe recorre aos métodos auxiliares `base64` para codificar e descodificar na *base-64*; e `sha1` para calcular o *hash* de acordo com o algoritmo criptográfico *sha1 (Secure Hash Algorithm 1)*.

4.2.5. Clientes Web

Para testar o servidor foram criados dois clientes em HTML/ *JavaScript* que utilizam a API *websocket* para se ligarem ao servidor. São definidas operações para os quatro eventos que estão disponíveis nos objetos *Websocket* (`onopen`, `onclose`, `onerror` e `onmessage`); e são utilizados os métodos `send()` e `close()` para enviar mensagens de texto e terminar a ligação, respetivamente.

Em ambos os clientes é possível encontrar o estado da ligação, o número total de mensagens recebidas, o tempo desde o início da ligação e um botão para ligar ou desligar a conexão com o servidor.

O primeiro simula um sistema de troca de mensagens, em versão *broadcast*, ou seja, permite o envio de mensagens, através de um formulário, de um cliente para todos os outros, tal como ilustra a Figura 4.5.

Figura 4.5: Cliente *websocket* - troca de mensagens

[18] https://code.google.com/p/rapidjson/

No segundo cliente é possível visualizar todas as mensagens que chegam ao SDW a partir do *ServerDeal*. É utilizado o método `JSON.parse()` para desserializar as mensagens em formato JSON que chegam do SDW. A Figura 4.6 representa a captura de ecrã num momento em que o cliente recebe mensagens. Neste exemplo, o cliente estava ligado ao servidor há quase 3 minutos e tinha recebido 22238 mensagens.

Figura 4.6: Cliente *websocket* que recebe mensagens do *ServerDeal*

4.3. Protocolos e formatos auxiliares

No desenvolvimento do protótipo foi utilizado o protocolo FIX, na comunicação entre o *ServerDeal* e o SDW, e o formato de mensagens em JSON entre o SDW e os clientes (navegadores Web), tal como se pode confirmar na Figura 4.1.

4.3.1. Protocolo FIX

O protocolo FIX define um conjunto de especificações para mensagens electrónicas e um modelo de comunicação para os mercados financeiros (Shaik, 2010). Foi desenvolvido através da colaboração de bancos, corretoras, bolsas, investidores institucionais e fornecedores de tecnologia de informação de todo o mundo que necessitam de uma linguagem comum para a negociação automática de instrumentos

financeiros. É utilizado, por exemplo, para a comunicação de cotações de valores mobiliários, ordens de compra ou venda, gestão de clientes, carteiras de títulos, submissão de pedidos, mudanças de ordens, relatórios de execução, entre outros.

Cada mensagem FIX contém três partes: cabeçalho, parte principal e rodapé, e pode ser representada como uma coleção de campos. Cada um dos campos é um par *<etiqueta> = <valor>*. Todos os campos terminam com o carácter delimitador ASCII "SOH" (*Start of Header*), geralmente representado por "^", "|" ou "<soh>". As etiquetas estão predefinidas por um número que corresponde a um determinado campo e os valores podem ser do tipo *int, float, char, time, date, data, string*. Todas as mensagens começam com "8=FIX.x.y", que indica a versão do protocolo e terminam com "10=nnn<SOH>", onde "nnn" representa a soma dos valores binários da mensagem (Onix Solutions Limited, 2011).

Exemplo de uma mensagem FIX e significado de cada um dos campos:

"8=FIX.4.1 ^ 9=0235 ^ 35=D ^ 34=10 ^ 43=N ^ 49=VENDOR ^ 50=CUSTOMER ^ 56=BROKER ^ 52=19980930-09:25:58 ^ 1=XQCCFUND ^ 11=10 ^ 21=1 ^ 55=EK ^ 48=277461109 ^ 22=1 ^ 54=1 ^ 38=10000 ^ 40=2 ^ 44=76.750000 ^ 10=165"

Versão do protoloco utilizado (8) = 4.1

Tamanho do corpo da mensagem (9) = 0235

Tipo da mensagem (35) = D (nova ordem)

Número de sequência da mensagem (34) = 10

Retransmissão de mensagem com este número de sequência (43) = N (no)

Emissor — identificação da empresa que envia a mensagem (49) = VENDOR

Emissor — secção dentro da empresa (50) = CUSTOMER

Receptor — identificação da empresa que recebe a mensagem (56) = BROKER

Tempo de transmissão da mensagem (52) = 19980930-09:25:58

Número de conta acordado entre o emissor e o receptor (1) = XQCCFUND

Identificação da ordem do emissor (11) = 10

Instruções de procedimentos para o receptor (21) = 1 (ordem automática, sem intervenção do intermediário financeiro)

Identificação do valor mobiliário (55) = EK

Identificador de segurança alternativo (48) = 277461109

Tipo de operação (54) = 1 (comprar)

Quantidade de ordens (38) = 10000

Tipo de ordem (40) = 2 (limite)

Preço por cada ordem (44) = 76.750000

Utilizado para verificação de integridade do conteúdo da mensagem(10) = 165

4.3.2. Formato JSON

JavaScript Object Notation (JSON) é um formato leve, de texto e independente da linguagem para a serialização e troca de dados estruturados. Está descrito no RFC 4627[19]. É baseado num subconjunto da linguagem de programação *JavaScript Standard ECMA-262*. JSON define um conjunto pequeno de regras de formatação para a representação portável de dados estruturados. As linguagens suportadas são *ActionScript, C/C++, C#, ColdFusion, Java, JavaScript, OCaml, Perl, PHP, ASP 3.0, Python, Rebol, Ruby, Lua, Progress*.

Podem ser representados quatro tipos primitivos: *strings, numbers, booleans* e *null; e* dois tipos estruturados: *objects* e *arrays* (Ecma International, 2013).

String — é uma sequência de zero ou mais caracteres *Unicode*, envolvido entre aspas duplas usando barras invertidas como carácter de *escape*;

Object — é uma coleção desordenada de zero ou mais pares nome/valor, onde o nome é uma *string* e o valor é do tipo *string*, *number*, *boolean*, *null*, *object* ou *array*; começa com "{" e termina com "}". Cada nome é seguido por ":" e os pares nome/valor são seguidos por ",";

Array — é uma sequência ordenada de zero ou mais valores; o *array* começa com "[" e termina com "]". Os valores são separados por ",".

Espaços em branco podem ser inseridos em qualquer parte dos símbolos.

A Listagem 4.1 representa um exemplo de uma mensagem no formato JSON. Esta mensagem refere-se à cotação da empresa "*Airbus Group NV*", da bolsa de valores de Paris ("XPAR"). Está indicado o tipo de cotação ("*CS*") e um conjunto de campos associados ao mesmo título.

```
{ "Symbol" : "AIR",
 "SecType": "CS",
 "SecExchange": "XPAR",
 "Fields":
   [
         {"Field":"0", "Value": "126"},
         {"Field":"3", "Value": "117"},
         {"Field":"2", "Value": "28.160"},
         {"Field":"1", "Value": "28.150"}
   ]
}
```

Listagem 4.1: Mensagem no formato JSON

[19] *www.ietf.org/rfc/rfc4627.txt*

4.4. Execução

Para testar a execução do protótipo é necessário, em primeiro lugar, iniciar o simulador de mercados financeiros. Este simulador disponibiliza cotações de títulos das bolsas de valores de Lisboa, Paris, Nova Iorque e NASDAQ.

Em segundo lugar, arrancar o *ServerDeal* que recebe dados do simulador. A versão que está a ser utilizada é adaptada da atual versão da plataforma Sifox.

Em terceiro lugar, arrancar o SDW que recebe as cotações no formato FIX do *ServerDeal*. Está à escuta de novos clientes na porta 8085. É possível visualizar na consola algumas mensagens produzidas pelo servidor, tais como: a quantidade de clientes que estão ligados a cada momento, sempre que ocorre algum erro ou algum cliente se desliga.

Finalmente, a partir de algum navegador Web, podem ser iniciados os clientes Web desenvolvidos de forma a trocarem mensagens entre si ou receberem cotações do *ServerDeal*.

Na Figura 4.7 é possível identificar as várias fases de uma ligação *websocket* entre um servidor (IP = 192.168.35.46) e um cliente (IP = 192.168.35.150). Foi utilizado o *Wireshark* para recolher estes dados.

No.	Time	Source	Destination	Protocol	Length	Info
128	12.4493310	192.168.35.150	192.168.35.46	TCP	78	49254 > 8085 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=1
129	12.4494030	192.168.35.46	192.168.35.150	TCP	74	8085 > 49254 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS
130	12.4496550	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=1 Ack=1 Win=132288 Len=0 TSval=
131	12.4505460	192.168.35.150	192.168.35.46	HTTP	508	GET / HTTP/1.1
132	12.4510750	192.168.35.46	192.168.35.150	HTTP	195	HTTP/1.1 101 Switching Protocols
133	12.4514000	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=130 Win=132144 Len=0 TS
134	12.4907270	192.168.35.46	192.168.35.150	WebSocket	241	WebSocket Text [FIN]
135	12.4912020	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=305 Win=131984 Len=0 TS
136	12.4913790	192.168.35.46	192.168.35.150	WebSocket	192	WebSocket Text [FIN]
137	12.4917050	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=431 Win=131856 Len=0 TS
138	12.4917390	192.168.35.46	192.168.35.150	WebSocket	252	WebSocket Text [FIN]
139	12.4921470	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=617 Win=131664 Len=0 TS
140	12.4921700	192.168.35.46	192.168.35.150	WebSocket	382	WebSocket Text [FIN]
141	12.4925050	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=933 Win=131344 Len=0 TS
...						
211	13.3401200	192.168.35.46	192.168.35.150	WebSocket	245	WebSocket Text [FIN]
212	13.3403940	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=8334 Win=130880 Len=0 TS
213	13.3404040	192.168.35.46	192.168.35.150	WebSocket	559	WebSocket Text [FIN]
214	13.3407570	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=8827 Win=130576 Len=0 TS
215	13.3407660	192.168.35.46	192.168.35.150	WebSocket	557	WebSocket Text [FIN]
216	13.3411590	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=9318 Win=130576 Len=0 TS
217	13.3411920	192.168.35.46	192.168.35.150	WebSocket	193	WebSocket Text [FIN]
218	13.3414700	192.168.35.150	192.168.35.46	TCP	66	49254 > 8085 [ACK] Seq=443 Ack=9445 Win=130944 Len=0 TS
219	13.3531180	192.168.35.150	192.168.35.46	WebSocket	72	WebSocket Connection Close [FIN] [MASKED][Malformed Pac
220	13.3555960	192.168.35.46	192.168.35.150	TCP	54	8085 > 49254 [RST, ACK] Seq=9445 Ack=449 Win=0 Len=0

Figura 4.7: Ligação *websocket* entre cliente e servidor

Primeiro ocorre a criação de um canal TCP com a troca de 3 pacotes. Em seguida o cliente envia um pedido inicial de atualização de protocolo e o servidor responde com o código 101 (*Switching Protocolos*). A partir deste momento o servidor envia consecutivamente mensagens para o cliente, que vai confirmando a chegada destas

com o envio de pacotes ACK. Para terminar a ligação o cliente envia uma mensagem do tipo *Close* para o servidor, que termina a ligação.

A mesma troca de mensagens pode ser visualizada, noutro formato, na Figura 4.8.

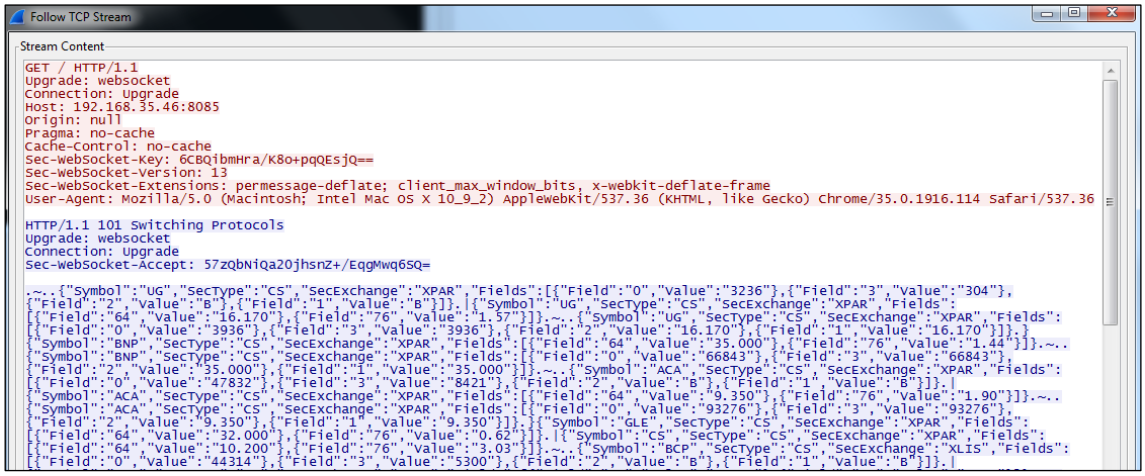


Figura 4.8: Início de ligação *websocket* entre cliente e servidor

Neste capítulo foi apresentada a tecnologia escolhida para elaborar o protótipo desenvolvido, a partir da comparação das alternativas apresentadas e da análise de vários estudos comparativos. Em seguida foi abordada a arquitetura do software, a organização do código e a descrição das principais classes implementadas. É ainda apresentado o protocolo FIX e o formato JSON, e o processo para executar o protótipo.

5. Avaliação e testes de performance

Este capítulo tem como objetivo avaliar o impacto da utilização do protocolo *websocket* na comunicação entre o SDW e os clientes Web. São apresentados dois conjuntos de testes de performance ao protótipo. Os primeiros comparam o SDW que é utilizado atualmente pela Finantech, com o protótipo. Os segundos testes pretendem avaliar a performance, escalabilidade e funcionamento do protótipo que foi desenvolvido ao longo do estágio.

5.1. SDW vs protótipo

A versão do SDW atualmente utilizada na Plataforma Sifox demorou alguns anos a ser desenvolvida. Este servidor efetua várias verificações e ligações a bases de dados, subscrições individuais de cotações de valores mobiliários para cada cliente, ligações aos serviços de *Alertas* e *Risco*, acesso aos dados de carteira dos clientes, entre muitas outras operações, tal como foram descritas em *1.3*.

Porém, o protótipo desenvolvido durante o estágio é bastante mais simples e limitado em termos de funcionalidades.

Por esta razão a comparação entre o SDW e o protótipo é condicionada pelas enormes diferenças entre ambos os serviços.

Para os testes de comparação entre a versão atual do SDW e o protótipo foi elaborada uma estrutura que está representada na Figura 5.1. Nesta arquitetura existe um *simulador de mercados financeiros* das bolsas de valores de Lisboa, Paris, Nova Iorque e NASDAQ, que fornece dados de mercado ao *ServerDeal*.

O *SDW1* e *Cliente1* representam as versões atuais do *SDW* e de um cliente Web fictício. De salientar que depois do *Cliente1* se ligar ao *SDW1*, aquele permanece apenas a receber as atualizações de um conjunto definido de títulos subscritos, não fazendo uso de outras funcionalidades.

O *SDW2* e o *Cliente2* representam o servidor e o cliente Web desenvolvidos para o protótipo, apresentados no capítulo anterior.

Dadas as diferenças em termos de complexidade entre o *SDW1* e o *SDW2*, apenas são comparados os dois canais de comunicação *a* e *b* da Figura 5.1.

Para realizar os testes é garantido que o *SDW1* e o *SDW2* recebem dados de mercado do *ServerDeal* em condições semelhantes. Ou seja, o tipo de ligação, a frequência do envio das informações e o tamanho de cada mensagem são iguais.

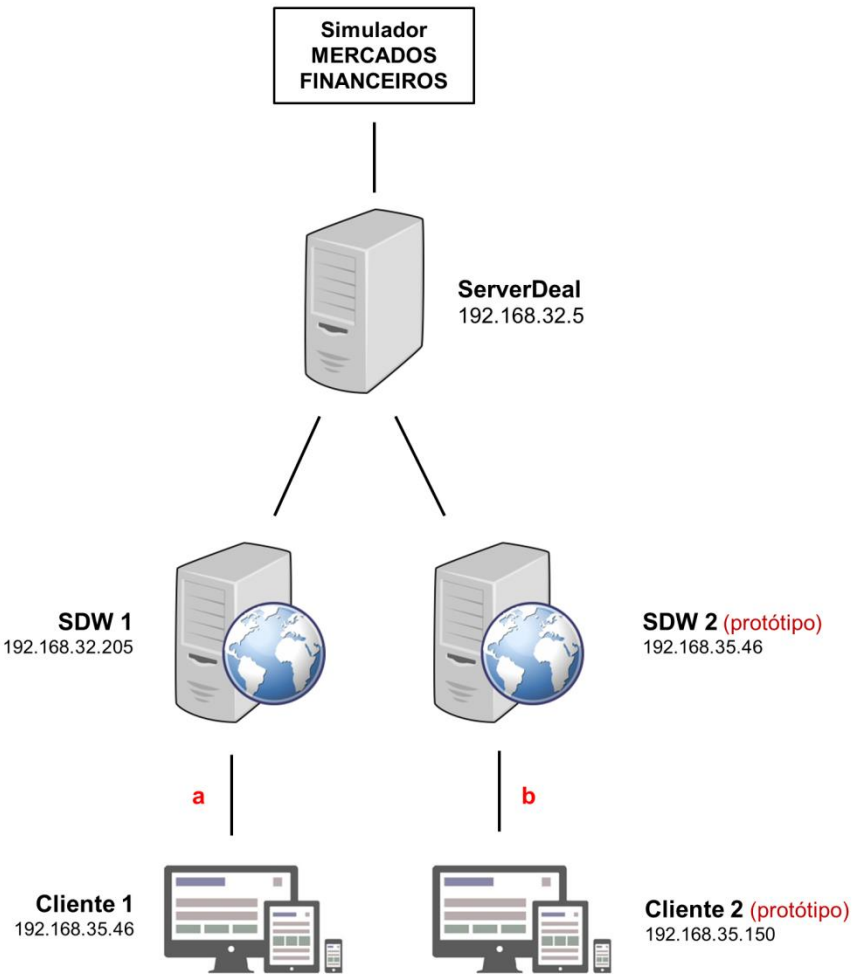


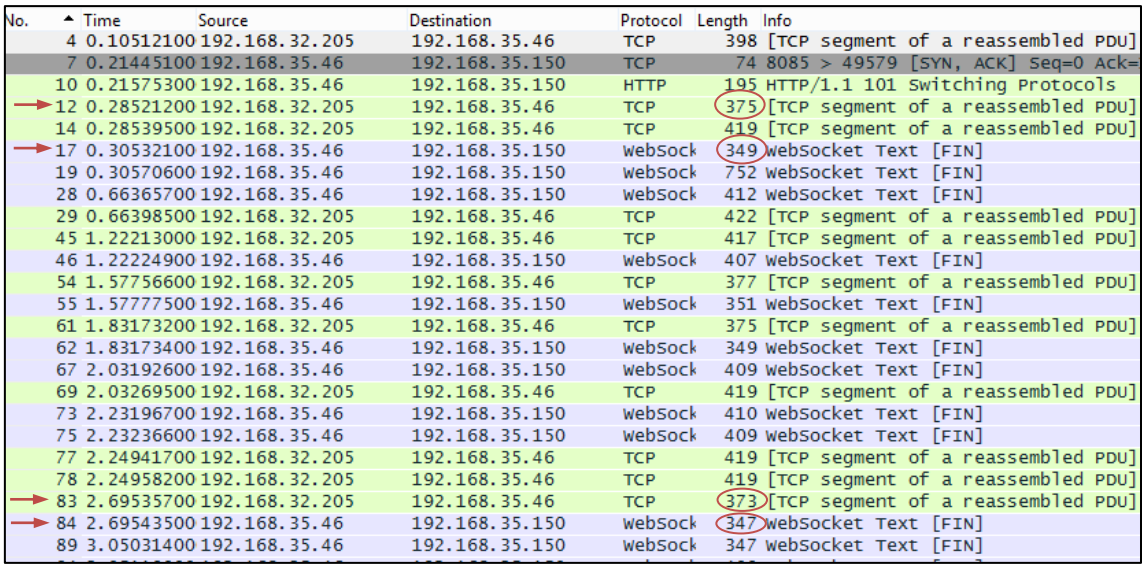
Figura 5.1: Arquitetura para os testes com SDW

Através do programa *Wireshark* é possível analisar o tráfego dos canais *a* e *b*. Foi definido o filtro: `(ip.src == 192.168.32.205 && ip.dst == 192.168.35.46) || (ip.src == 192.168.35.46 && ip.dst == 192.168.35.150)`, de forma a visualizar apenas as mensagens que seguem de *SDW1* para *Cliente1* e de *SDW2* para *Cliente2*. A Figura 5.2 apresenta uma captura de ecrã num instante em que a comunicação está a decorrer.

Ao analisar o fluxo de informação em ambos os canais de comunicação destacam-se dois pontos: um relacionado com a utilização da largura de banda, outro com o número total de troca de mensagens.

Em primeiro lugar, comparando duas mensagens com o mesmo conteúdo em canais de comunicação diferente, verifica-se que o tamanho das mensagens no canal *b* é menor que no canal *a*. Por exemplo, a mensagem número 12 — Figura 5.2 — com as últimas cotações do título “TWTR” (*Twitter, Inc.*), no canal *b* ocupa 349 *bytes*, enquanto no canal *a* (mensagem número 17), ocupa 375 *bytes*. Isto representa uma diminuição de 7,45% na utilização da largura de banda. Ou as mensagens números 83

e 84 que representam a cotação do mesmo valor mobiliário ("*Covidien PLC*"), também apresentam uma diferença percentual de 7,49%.



No.	Time	Source	Destination	Protocol	Length	Info
4	0.10512100	192.168.32.205	192.168.35.46	TCP	398	[TCP segment of a reassembled PDU]
7	0.21445100	192.168.35.46	192.168.35.150	TCP	74	8085 > 49579 [SYN, ACK] Seq=0 Ack=
10	0.21575300	192.168.35.46	192.168.35.150	HTTP	195	HTTP/1.1 101 Switching Protocols
12	0.28521200	192.168.32.205	192.168.35.46	TCP	375	[TCP segment of a reassembled PDU]
14	0.28539500	192.168.32.205	192.168.35.46	TCP	419	[TCP segment of a reassembled PDU]
17	0.30532100	192.168.35.46	192.168.35.150	WebSock	349	WebSocket Text [FIN]
19	0.30570600	192.168.35.46	192.168.35.150	WebSock	752	WebSocket Text [FIN]
28	0.66365700	192.168.35.46	192.168.35.150	WebSock	412	WebSocket Text [FIN]
29	0.66398500	192.168.32.205	192.168.35.46	TCP	422	[TCP segment of a reassembled PDU]
45	1.22213000	192.168.32.205	192.168.35.46	TCP	417	[TCP segment of a reassembled PDU]
46	1.22224900	192.168.35.46	192.168.35.150	WebSock	407	WebSocket Text [FIN]
54	1.57756600	192.168.32.205	192.168.35.46	TCP	377	[TCP segment of a reassembled PDU]
55	1.57777500	192.168.35.46	192.168.35.150	WebSock	351	WebSocket Text [FIN]
61	1.83173200	192.168.32.205	192.168.35.46	TCP	375	[TCP segment of a reassembled PDU]
62	1.83173400	192.168.35.46	192.168.35.150	WebSock	349	WebSocket Text [FIN]
67	2.03192600	192.168.35.46	192.168.35.150	WebSock	409	WebSocket Text [FIN]
69	2.03269500	192.168.32.205	192.168.35.46	TCP	419	[TCP segment of a reassembled PDU]
73	2.23196700	192.168.35.46	192.168.35.150	WebSock	410	WebSocket Text [FIN]
75	2.23236600	192.168.35.46	192.168.35.150	WebSock	409	WebSocket Text [FIN]
77	2.24941700	192.168.32.205	192.168.35.46	TCP	419	[TCP segment of a reassembled PDU]
78	2.24958200	192.168.32.205	192.168.35.46	TCP	419	[TCP segment of a reassembled PDU]
83	2.69535700	192.168.32.205	192.168.35.46	TCP	373	[TCP segment of a reassembled PDU]
84	2.69543500	192.168.35.46	192.168.35.150	WebSock	347	WebSocket Text [FIN]
89	3.05031400	192.168.35.46	192.168.35.150	WebSock	347	WebSocket Text [FIN]

Figura 5.2: Comparação entre mensagens nos canais a e b, dos servidores para clientes

Em segundo lugar, é possível confirmar que um canal *websocket* permite agregar vários pacotes numa única mensagem[20]. Por exemplo, a mensagem número 172 — destacada na Figura 5.4 —, contém a informação de três cotações diferentes da bolsa de Paris: *L'Oreal SA, Alstom SA* e *Capgemini*. Isto significa que o protocolo *websocket* permite trocar um menor número de mensagens para transportar a mesma quantidade de informação.

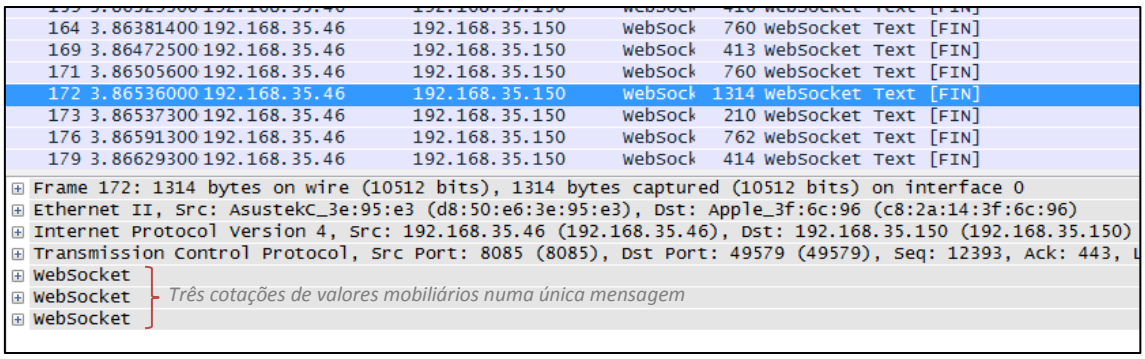


Figura 5.3: Uma mensagem *websocket* com três pacotes

Ainda foram realizados mais alguns testes. Porém, as diferenças entre ambos os sistemas, *SDW1* e *SDW2*, não permitem retirar mais conclusões. Por exemplo, a

[20] RFC 6455 — *The WebSocket Protocol* — secção 1.2: *After a successful handshake, clients and servers transfer data back and forth in conceptual units referred to in this specification as "messages". On the wire, a message is composed of one or more frames. The WebSocket message does not necessarily correspond to a particular network layer framing, as a fragmented message may be coalesced or split by an intermediary. A frame has an associated type. Each frame belonging to the same message contains the same type of data.*

Tabela 5.1 e Figura 5.4 representam o tráfego gerado nos canais *a* e *b* durante uma ligação com a duração de 60 minutos. Contudo, o *SDW1* implementa várias técnicas para selecionar e otimizar o envio de mensagens para o *Cliente1*, o que resulta numa diminuição do número de mensagens enviadas.

Tabela 5.1: Tráfego gerado nos canais a e b, durante 60 minutos

Canal	Nº total mensagens	Média mensagens p/ segundo	Tamanho médio mensagens (bytes)	Tráfego (Kb)
a	23571	5,5	359	8475
b	32370	9	491	15910

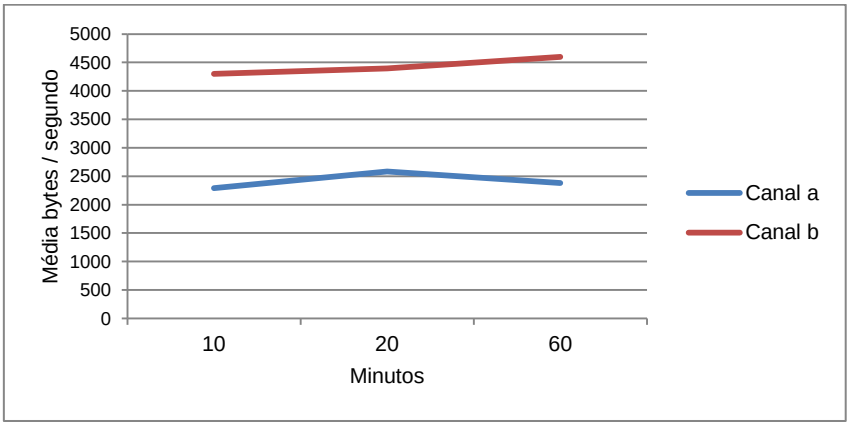


Figura 5.4: Quantidade de tráfego gerado em média, por segundo, nos canais a e b

5.2. Protótipo

Neste segundo conjunto de testes pretende-se avaliar a performance, escalabilidade e robustez do protótipo. Foram utilizados vários programas para recolher os dados, nomeadamente, *Process Explorer (Windows Sysinternals), PushToTest, Selenium IDE e NeoLoad (NEOTYS)*. O computador usado tem um processador Intel de 3,3 GHz com 2 núcleos (4 *threads*); 4 GB de memória RAM e 300 GB de disco rígido.

A partir da Figura 5.5 é possível analisar os recursos gastos pelo servidor em termos de memória. Neste teste, durante o tempo em que são ligados 400 clientes em simultâneo ao servidor, a receberem continuamente os mesmos dados de mercado, a quantidade de memória física e virtual necessária é registada. Verifica-se que a memória reservada se mantém relativamente estável, com um crescimento linear, enquanto o número de clientes aumenta exponencialmente.

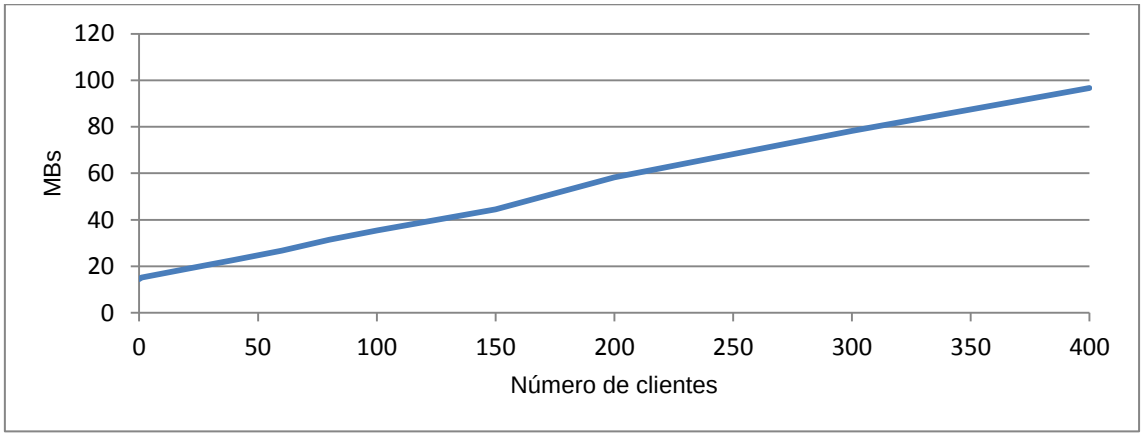


Figura 5.5: Monitorização da memória física e virtual durante a ligação de 400 clientes

Em termos de utilização de processador verificam-se ligeiras oscilações enquanto as ligações estão a ser criadas. Porém, a partir do momento em que se estabelecem as 400 novas ligações, o processo onde está a correr o servidor raramente ultrapassa os 3% de utilização do processador.

De salientar que, atualmente, o cliente da Finantech com mais utilizadores do SDW não deve ultrapassar as 100 ligações em simultâneo.

A partir da Tabela 5.2 e da Figura 5.6 é possível confirmar que a gestão de *threads* é corretamente executada, quer na criação de novas ligações, quer no encerramento das mesmas.

Tabela 5.2: Número de *threads* criados para 0, 1 e 2 ligações

Clientes	0	1	2
Threads	6	8	10

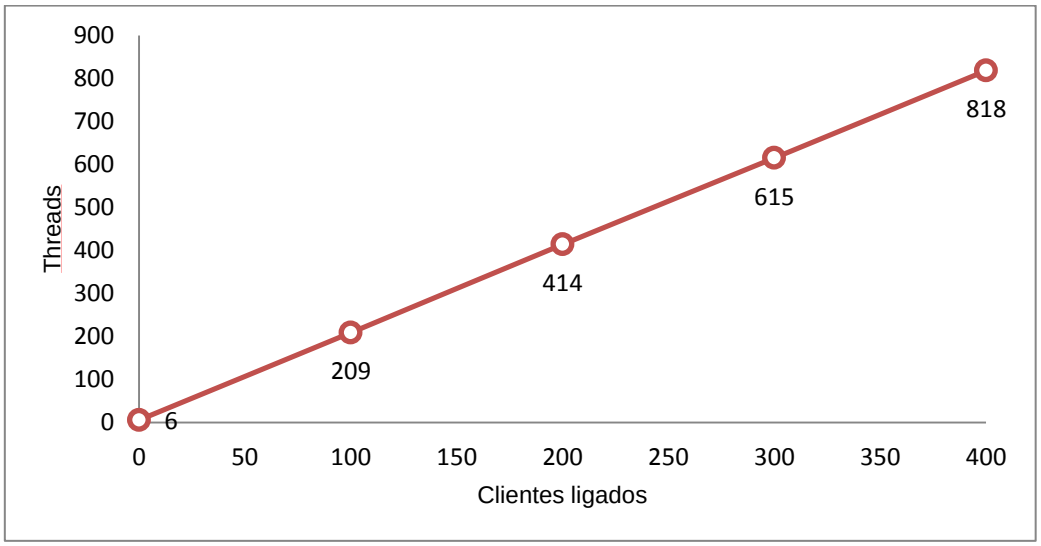


Figura 5.6: Relação entre o número de clientes ligados e a quantidade de *threads*

Finalmente são executados um conjunto de testes para confirmar a robustez do servidor, nomeadamente, quanto ao tratamento de erros, na criação e encerramento abrupto de um conjunto grande de clientes, diversificando a origem dos clientes quanto ao endereço IP e plataforma digital utilizada (Pc, iMac, *tablet* e telemóvel). Em todos os casos são emitidos avisos no caso da ocorrência de algum erro; a manutenção e atualização de todos os clientes é mantida permanentemente; e a abertura e o fecho de ligações é assegurada independentemente da sua origem.

6. Conclusão

As transformações ocorridas na Web, que a tornam uma plataforma para aplicações distribuídas e altamente dinâmicas, obrigam o aparecimento de alternativas ao modelo original de pedido/ resposta do HTTP. Tal como defende (Lubbers P. A., 2010), o protocolo HTTP não foi desenhado para comunicações bidirecionais, assíncronas e em tempo real.

Atualmente, os serviços e as aplicações Web apresentam-se como sistemas cada vez mais complexos. Estas plataformas exigem uma utilização eficiente da largura de banda, soluções escaláveis, comunicação bidirecional entre servidor e clientes, disponibilidade em múltiplas plataformas e linguagens, entre outros requisitos.

Neste relatório são inicialmente apresentadas alternativas tecnológicas para a criação de canais de comunicação para a Web, entre cliente e servidor.

Das várias soluções apresentadas destaca-se o protocolo *websocket*. Esta tecnologia permite a criação de uma ligação permanente, bidirecional, em tempo real e compatível com o HTML. Pelo facto de dispensar a utilização de cabeçalhos HTTP depois de estabelecida a ligação e usar um formato de mensagens simples e leve, o protocolo *websocket* permite uma eficiente utilização da largura de banda. A API *websocket* fornece uma interface simples e intuitiva para utilizar o protocolo.

É também possível desenvolver aplicações Web seguras, baseadas em *websocket*. Ou seja, aplicações capazes de disponibilizar serviços que garantem confidencialidade, integridade, disponibilidade e autenticidade, a clientes e servidores. O conjunto de recomendações práticas que são apresentadas asseguram que, mesmo com a utilização de *websocket*, é possível proteger todos os intervenientes da maioria dos ataques informáticos atuais.

Neste trabalho foi apresentado a arquitetura de um protótipo de um sistema para melhorar a comunicação de dados de mercado, notícias e pedidos de compra e venda de valores mobiliários, entre o *ServerDeal* e as aplicações cliente *SifoxDealWeb* e *SifoxDealMobile.* Ou seja, para diminuir a latência na comunicação e aumentar a coerência de dados em ambas as partes.

Os testes de comparação entre o SDW atual e o protótipo demonstram que o uso do protocolo *websocket* permite uma utilização mais eficiente da largura de banda e um menor número de troca de mensagens para a mesma quantidade de informação. Através da análise da utilização do processador, memória física e virtual, *gestão de threads* e tratamento de erros, confirma-se a performance, escalabilidade, confiança e o correto funcionamento do protótipo, mesmo quando sujeito a um elevado número de ligações de clientes.

Em suma, o trabalho desenvolvido ao longo do estágio demonstra que a utilização do protocolo *websocket* nas comunicações com o SDW é benéfica para a Plataforma Sifox. Por um lado, é garantida uma melhor performance na transmissão de dados, com a diminuição da latência e o aumento na coerência de dados entre o SDW e respetivos clientes. Por outro lado, a arquitetura geral do SDW é simplificada, permitindo ficar independente do IIS e ser apenas necessário a criação de um canal de comunicação para cada cliente.

Porém, o protótipo descrito neste relatório apresenta fragilidades. A principal prende-se com facto de ser um protótipo que simula um sistema bastante mais simples e com menos funcionalidades, comparativamente ao SDW atual. Este facto limita os termos de comparação entre ambos os serviços, não apenas para testar a performance do servidor, mas também dos clientes.

Outra limitação está relacionada com o nível de segurança da ligação entre o SDW e os clientes Web. Por exemplo, não foi utilizado o esquema *wss* para as ligações, não existe qualquer verificação no servidor da origem do cliente ou do subprotocolo, nem existem restrições quanto ao número de novas ligações que podem ser aceites, entre outras medidas de segurança.

Finalmente, o facto dos clientes do protótipo que se ligam ao SDW serem todos iguais também representa uma limitação. O que acontece na realidade é que cada cliente assume um comportamento diferente de todos os outros, nomeadamente, quanto ao conjunto de títulos que subscreve ou operações que efetua.

Após a realização deste trabalho é sugerida a atualização do protocolo *websocket* e da API. A primeira está relacionada com a gestão de ligações *websocket* abertas e que não estão a passar dados. Atualmente o protocolo não prevê quaisquer mecanismos de *keep-alive*, tirando o facto de que especifica mensagens do tipo *ping* e *pong*. O que implica que a manutenção destas ligações tem de ser feita manualmente pelo programador do servidor.

A segunda melhoria aqui proposta é no método `send(data)` da API *websocket*. Atualmente o método `send()` recebe apenas um argumento, com a informação que se pretende enviar no formato texto ou dados binários. E este método tenta enviar toda a informação de uma só vez. Seria útil se este método aceitasse mais argumentos e fosse possível controlar o envio da informação, por exemplo, consoante o tráfego na rede, a largura de banda disponível ou tamanho da mensagem que se pretende enviar. Uma vez mais, este tipo de gestão do envio da informação tem de ser feito manualmente pelo programador.

A longo prazo pretende-se transformar este protótipo numa biblioteca DLL, portável para ambientes *Windows* e *Linux,* que possa ser incluída em qualquer projeto e que seja capaz de atuar como cliente e servidor, implementando o protocolo *websocket*

para a comunicação. Esta biblioteca seria apenas responsável por estabelecer as ligações, desconhecendo o conteúdo das mensagens que são trocadas. Por exemplo, poderia servir para cotações, processos de autenticação e autorização de utilizadores, pesquisas em bases de dados, acesso a serviços da Plataforma Sifox, entre outros.

No futuro também será necessário alterar todos os componentes que se ligam ao SDW (Figura 1.3). Por exemplo, será importante desenvolver um cliente em C++, capaz de se ligar ao servidor de forma a disponibilizar um vasto conjunto de funcionalidades, nomeadamente, autenticação, subscrições e operações de compra e venda.

7. Referências bibliográficas

Agarwal, S. (2012). Real-time Web Application Roadblock: Performance Penalty of HTML Sockets. *NEC Europe Laboratories*.

APB. (2014). *Associação Portuguesa de Bancos (APB)*. Obtido em 15 de Abril de 2014, de www.apb.pt

Bassbouss, L., Tritschler, M., Steglich, S., Tanaka, K., & Miyazaki, Y. (2013). Towards a Multi-Screen Application Model for the Web. *IEEE 37th Annual Computer Software and Applications Conference Workshops*, (pp. 528-533).

BM&F Bovespa. (2007). *Bolsa de Valores de São Paulo (BM&F Bovespa).* Obtido em 7 de Abril de 2014, de www.bmaiscompet.com.br/arquivos/MercadodeCapitaisBovespa.pdf

Bozdag, E., Mesbah, A., & Deursen, A. (2009). Performance Testing of Data Delivery Techniques for AJAX Applications. *J. Web Eng. 8, 4* , 287-315.

Carreira, P. J. (2008). *Engenharia de Software.* Obtido em 16 de Abril de 2014, de http://disciplinas.ist.utl.pt/~leic-es.daemon/2008-2009/labs/stripes-tutorial/

Cassetti, O., & Luz, S. (2011). The WebSocket API as supporting technology for distributed and agent-driven data mining. *Next Generation Data Mining Summit*.

CMVM. (2011). *Comissão do Mercado e Valores Mobiliários (CMVM).* Obtido em 30 de Março de 2014, de www.cmvm.pt/CMVM/Publicacoes/Guia/Pages/indice_guia.aspx

Collina, M., Corazza, G. E., & Vanelli-Coralli, A. (2012). Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. *23rd Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, (pp. 36-41).

Corcoran, P., Mooney, P., Winstanley, A., & Bertolotto, M. (2011). Effective Vector Data Transmission and Visualization Using HTML5. *GISRUK 2011 Conference.* Portsmouth, UK.

Cravens, J. (21 de Abril de 2013). *Push Notifications to the Browser With Server Sent Events*. Obtido em 27 de Abril de 2014, de HTML5 Hacks: http://html5hacks.com/blog/2013/04/21/push-notifications-to-the-browser-with-server-sent-events/

Dierks, T., & Rescorla, E. (2008). *RFC 5246 - The Transport Layer Security (TLS).* Internet Engineering Task Force.

Ecma International. (10 de 2013). *The JSON Data Interchange Format .* Obtido em 3 de Junho de 2014, de http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

Erkkilä, J.-P. (2013). *WebSocket Security Analysis*. Obtido em 25 de Maio de 2014, de Aalto University School of Science: http://juerkkil.iki.fi/files/writings/Websocket2012.pdf

Euronext. (2013). *Euronext Lisboa (Euronext).* Obtido em 30 de Março de 2014, de www.bolsadelisboa.com.pt/centro-de-aprendizagem

Fan, B., & Wang, T. (2010). Message Broker Using Asynchronous Method Invocation in Web Service and Its Evaluation. *Software Testing, Verification, and Validation Workshops (ICSTW)*, (pp. 265-273).

Felício, M. V. (2012). *LightStream – Sistema de comunicação Publish/Subscribe com WebSockets.* Trabalho Final de Mestrado, Instituto Superior Técnico de Lisboa, Engenharia de Electrónica e Telecomunicações e de Computadores, Lisboa.

Furukawa, Y. (2012). Web-based Control Application using Websocket. *International Conference on Accelerator and Large Experimental Physics Control Systems*, (pp. 695-697). France.

Garrett, J. J. (2005). *Ajax: A New Approach to Web Applications*. Obtido em 20 de Março de 2014, de www.adaptivepath.com/ideas/ajax-new-approach-web-applications

Goncalves, C. B. (1984). *Casa da Moeda do Brasil: 290 anos de História, 1694/1984.* Rio de Janeiro: Imprinta.

Google Project Hosting. (2011). *Pubsubhubbub - Project Home*. Obtido em 3 de Maio de 2014, de https://code.google.com/p/pubsubhubbub/

Govaerts, S., Verbert, K., Dahrendorf, D., Ullrich, C., Schmidt, M., Werkle, M., et al. (2011). *Towards Responsive Open Learning Environments: The ROLE Interoperability Framework* (Vol. 6964). Springer Berlin Heidelberg.

Hanson, J. (2014). *What is WebRTC?* Obtido em 2 de Maio de 2014, de www.pubnub.com/blog/what-is-webrtc/

Heinrich, M., & Gaedke, M. (2012). Data Binding for Standard-based Web Applications. *ACM Symposium on Applied Computing*, 652-657.

Heroku Dev Center. (Abril de 2014). *Websocket security*. Obtido em 26 de Maio de 2014, de https://devcenter.heroku.com/articles/websocket-security

Hickson, A. (2011). *The WebSocket API*. Obtido em 10 de Abril de 2014, de http://www.w3.org/TR/2011/WD-websockets-20110419/

IBS. (2006). *ISCTE Business School (IBS).* Obtido em 11 de Abril de 2014, de http://ibs.iscte.pt/jogoinvestimento/mercados_financeiros.swf

Isak, C. (2013). *What is WebRTC? – Overview for Busy People*. Obtido em 23 de Abril de 2014, de www.telepresence24.com/?p=1804

Jõhvik, M. (2011). *Push-based versus pull-based data transfer in AJAX applications.* Bachelor's thesis (6EAP), University of Tartu, Computer Science.

Kapetanakis, K., Panagiotakis, S., & Malamos, A. G. (2013). HTML5 and WebSockets; challenges in network 3D collaboration. *17th Panhellenic Conference on Informatics*, 33-38.

Koren, I., Guth, A., & Klamma, R. (2013). Shared Editing on the Web: A Classification of Developer Support Libraries. *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, (pp. 468-477). Germany.

Kumar, A. (2014). *HTTP: The Fundamentals*. Obtido em 10 de Abril de 2014, de http://pypix.com/tools-and-tips/http-fundamentals/

Laine, M., & Säilä, K. (2012). *Performance Evaluation of XMPP on the Web.*

Lopes, J. L. (2012). *Suporte para Interface Web de Aplicações Distribuídas: Estudo da Tecnologia Websocket.* (U. F. Sul, Ed.) Obtido de http://saloon.inf.ufrgs.br/twiki-data/Disciplinas/CMP167/TFIFSULJoaoLopes/artigo-joao-tf-pod.pdf

Lubbers, P. A. (2010). *Pro HTML5 Programming: Powerful APIs for Richer Internet.* Nova Iorque: Apress.

Lubbers, P., & Greco, F. (2012). *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web*. Obtido em 18 de Abril de 2014, de www.websocket.org/quantum.html

Martins, M. M. (2007). *Macroeconomia I - LEC201.* Obtido em 30 de Março de 2014, de www.fep.up.pt/disciplinas/lec201/Textos/Cap1_slides.pdf

Moraes, T. F., Amorim, P. H., Silva, J. V., & Pedrini, H. (2012). Visualização Interativa em Tempo Real de Dados Médicos na Web. *IX Workshop de Realidade Virtual e Aumentada*, 1-6.

Onix Solutions Limited. (2011). *Financial Information Exchange protocol (FIX) Standards based messaging*. Obtido em 26 de Abril de 2014, de FIX 4.0 : Fields by Tag: www.onixs.biz/fix-dictionary/4.0/fields_by_tag.html

Pierro, G., Cavallari, .., Guida, S. D., & Innocente, V. (2010). Fast access to the CMS detector condition data employing HTML5 technologies. *Technical Report CMS-CR- 2010-222.* CERN, Geneva.

Pimentel, V., & Nickerson, B. G. (2012). Web Display of Real-time Wind Sensor Data. *Internet Computing, IEEE. 16*, pp. 45 - 53. Canada: IEEE.

Quintana, T. (2013). *HTML5 WebRTC and SIP Over WebSockets.* Obtido em 23 de Maio de 2014, de http://www.sipforum.org/component/option,com_docman/task,doc_view/gid,622/Itemid,261/

Qveflander, N. (2010). *Pushing real time data using html5 web sockets.* Master's thesis, Umea University, Department of Computing Science.

Ramos, L. B. (2013). *Um chat pictográfico para o SCALA (Sistema de Comunicação Alternativa para o Letramento de pessoas com Autismo).* Trabalho de graduação, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Porto Alegre.

Sanchez, F. (2011). *HTML 5: Início, Meio e Fim – Introdução – Parte 1*. Obtido em 20 de Abril de 2014, de http://fabriciosanchez.com.br/2/html-5-inicio-meio-e-fim-introducao-parte-1/

Santos, F. T. (2001). *A Evolução do Mercado de Capitais Português.* Obtido em 28 de Março de 2014, de www.cmvm.pt/CMVM/A%20CMVM/Conferencias/Intervencoes/Documents/58ee22e0dc804950ada7a5517f16cd1a200104EcoPura.pdf

Shaik, K. (2010). *FIX Protocol One Day Course.* Obtido em 16 de Abril de 2014, de www.ksvali.com

Sheiko, D. (2012). *WebSockets vs Server-Sent Events vs Long-polling*. Obtido em 2 de Maio de 2014, de http://dsheiko.com/weblog/websockets-vs-sse-vs-long-polling/

Silva, C. M. (2006). *Tecnologias de informação e comunicação e suas implicações para o exercício da nova cidadania.* Obtido em 29 de Março de 2014, de Pós-modernidade, política e educação: http://www.angelfire.com/sk/holgonsi/claudia.html

Smullen, C., & Smullen, S. (2008). An Eperimental Study of AJAX Application Performance. *Journal of Software*.

Swamy, R. N., & Mahadevan, D. G. (2011). Event Driven Architecture using HTML5 WebSockets for Wireless Sensor Networks. *Planetary Scientific Research Center.* White Papers.

Tableless. (2013). Obtido em 18 de Abril de 2014, de Guia de Referência sobre HTML5: http://tableless.com.br/html5/print.php?chapter=15

The Chromium Projects. (2013). *SPDY: An experimental protocol for a faster web*. Obtido em 2 de Maio de 2014, de www.chromium.org/spdy/spdy-whitepaper

Varela, T. D. (2011). *Implementação e análise da utilização de websockets em sistemas computacionais.* (U. L. Brasil, Ed.) Obtido em 10 de Junho de 2014, de http://www.ulbra.inf.br/joomla/images/documentos/TCCs/2011_01/TCCII_CC_Thiago Varela.pdf

Vieira, P. C. (2004). *Microeconomia I.* Obtido em 10 de Abril de 2014, de www.fep.up.pt/docentes/pcosme/MicroI/Aula2.pdf

Vinoski, S. (2012). Server-Sent Events with Yaws. *16*, pp. 98 - 102. IEEE Computer Society.

Wang, V., Salim, F., & Moskovits, P. (2013). *The Definitive Guide to HTML5 WebSocket.* New York: Apress.

Wessels, A., Purvis, M., Jackson, J., & Rahman, S. (. (2011). Remote Data Visualization through WebSockets. *Eighth International Conference on Information Technology: New Generations*, (pp. 1050-1051).

ZEPEDA, J. S., & CHAPA, S. V. (2007). From Desktop Applications Towards Ajax Web Applications. *4th International Conference* (pp. 193-196). Electrical and Electronics Engineering.